Increasing the Semantic Similarity of Object-Oriented Domain Models by Performing Behavioral Analysis First

by

Davor Svetinovic

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2006

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The main goal of any object-oriented analysis (OOA) method is to produce a model that aids in understanding and communicating knowledge about a modeled domain. A higher degree of similarity among independently produced domain models provides an indication of how well the domain was understood by the different analysts, i.e., more similar models indicate a closer and a more common understanding of a domain. A common understanding is of critical importance for effective knowledge communication and sharing.

The core of any OOA method is discovering and understanding concepts and their relationships in a domain. The main artifact produced by an OOA method is a domain model of the domain. A domain model often serves as the main source of design concepts during object-oriented design (OOD). This thesis evaluates two OOA methods by comparing the degree of similarity of the resulting domain models.

In particular, this work compares the semantic similarity of domain models extracted from use cases by

1. specification of sequence diagrams and then domain models, and

2. specification of unified use case statecharts and then domain models.

The thesis makes case studies out of the application of the first method to 31 instances of large Voice-over-IP (VoIP) system and its information management system (IMS) and to 3 small elevator systems, and out of the application of the second method to 46 instances of the same large VoIP system and its IMS and to 12 instances of a medium-sized elevator system.

From an analysis of data from these case studies, the thesis concludes that there is an increase of 10% in the semantic similarity of domain models produced using the second method, but at the cost of less than or equal to 25% more analysis time.

In Memory of

my Grandma, Rada Obućina-Lukić,

1914 – 1998

Acknowledgments

First of all, I'd like to thank my supervisors Prof. Daniel M. Berry and Dr. Michael W. Godfrey for all their support and guidance.

To Prof. Daniel M. Berry, I'd like to thank for all the advice, comments, critique, and the overall guidance in a direction of performing good quality research and the creation of this thesis as a result. The work with Dan has created in me almost an obsession with the importance of the quality and the significance of the research work tackled, insistance on a strong work and research ethics, and development of a unique management and teaching style. I had the opportunity to observe and learn an exceptional, complete, and very productive approach to research, work, teaching, and management, based on focus on quality, respect, and collaboration. Thanks for letting me pursue the topic of my choice and for all the help along the way. Thanks Dan!

To Dr. Michael W. Godfrey, first I'd like to thank for keeping me as his student after my Master's degree. The research relationship and the work we started during my Master's studies expanded and enriched even more during my Ph.D. studies. The underlying aspect of our work was insisting on and pursuing proper

scientific work, and not just keeping at what is basic but always pushing the barrier and exploring further and questioning everything along the way. Thanks for teaching me and improving my ability to think clearly, objectivelly, and value proper scientific criticism. Thanks for all the advice and tips on handling all the major obstacles in a Ph.D. student's life! Thanks Mike!

Thanks to Dr. Nancy A. Day, who became influential in the latter stages of my Ph.D. research, both as a Ph.D. committee member and as a person who supervised my teaching work. In particular, thanks for giving me the opportunity to teach my favorite course, and for numerous, concrete, and beneficial questions which helped tailor and improve my research and thesis. Thanks Nancy!

Thanks to my other Ph.D. committee members, Prof. Richard Holt, Dr. Ladan Tahvildari, and Prof. John Mylopoulos, for their time in reviewing this thesis work and for providing helpful comments and advice.

Thanks to Dr. Andrew Malton for numerous, interesting, and mostly quite conflicting discussions on many software engineering topics. It is simply amazing how much a good discussion with strong opposite views can make one reflect and improve one's own opinion!

Thanks to Dr. Joanne M. Atlee, who due to her positions happened to be a decision maker on a number of extremely important issues during my Ph.D. studies and as such greatly impacted my work!

Thanks to all the students that I worked with as a teaching assistant and as a course instructor. Also, thanks to all the profs and teaching assistants that I worked with. Thanks to all the staff of the university who helped with all administrative

issues.

Thanks to Xinyi Dong, Ahmed Hassan, Alan Grosskurth, Cory Kapser, Jingwei Wu, Lijie Zou, and all the other Software Architecture Group students with whom I shared office and lab space during these years and shared many opinions, complaints, and jokes. The same goes for Shahram Esmaeilsabzali, Vlad Ciubotariu, Jianwei Niu, Yun Lu, Yuan Peng, Alma L. Juarez Dominguez, and all other Waterloo Formal Methods group students with whom I shared the same when I used to be an unofficial WATFORM-er. Thanks to all other friends, including the Salsa and Muay Thai gang.

Finally, many thanks to my parents and my sister for their not just Ph.D., but all life-long love, care, and support!

This work was supported in part by Canadian NSERC Postgraduate Scholarship PGS B-255929-2002 and a Canadian FCAR Doctoral Scholarship.

Davor Svetinovic,

Sherbrooke, September 27, 2006.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In 1967, Ole-Johan Dahl and Kristen Nygaard presented the first object-oriented programming (OOP) language, Simula 67 [23]. In 1982, Grady Booch published his paper on object-oriented design (OOD) [13]. In 1988, Sally Shlaer and Stephen J. Mellor published their book on object-oriented analysis (OOA) [86].

These three events were major milestones in the development of the object-oriented (OO) paradigm of software development. Today, object orientation is not just one of the oldest software development paradigms, it is also one of the most widespread. From OOP languages to different OO modeling standards and frameworks, object orientation shapes the ways we think about business and software systems, how we organize our development processes, and so on.

Why did this happen?

It appears that the eventual widespread adoption of object orientation was fueled partially by the impact of Booch's 1982 paper "Object-Oriented Design" [13],

and in particular due to the two of his claims in that paper. His first claim, in Section 4.3.1, "Identify Objects and Their Attributes", is:

> *This step is a simple task; we will repeat our informal strategy in the abstract world, underlying [sic, should be "underlining"] the applicable nouns and adjectives....*

The second claim, in Section 4.3.2, "Identify Operations on the Objects", is:

> *This too is a simple task; we will repeat the above process, this time underlining the applicable verbs and adverbs....*

These two sections explain the main steps of an OOD method that Booch was advocating. Today, we know that these two steps form the foundation of OOA. We know also that these tasks are not that *simple* and, as stated, they are not sufficient for the production of semantically similar, high-quality OOA models.

Hatton [48], Kaindl [54], and Kramer [59] have recently indicated an urgent need for experimentation aimed at validating the effectiveness not just of object orientation but of all software engineering abstraction techniques and methods.

One might argue that from the requirements engineering (RE) perspective, *consistency* is one of the most important aspects of the high-quality analysis models [e.g., 65]. One particular form of *consistency* is across different groups of analysts independently analyzing the same domain, i.e., how consistent are the domain models (DMs) [1] of the *same* domain produced by independent groups of analysts using the same OOA method?

[1] This thesis uses the term "domain model (DM)" for what is known as "OOA class model" or "conceptual model" in OOA literature [e.g., 62].

To avoid confusing this type of consistency with other types of consistency, this thesis calls consistency of independently specified DMs *semantic similarity* of the independently specified DMs. Semantic similarity of independently specified DMs is a measure of the reproducibility and predictability of the results of the OOA methods. Given the same domain, the same OOA method should ideally produce the *same* DMs no matter who is performing the analysis.

1.1 Semantic Similarity

It is hard to define "semantic similarity" of DMs precisely. However, a domain expert *knows* when two models are semantically similar and when they are not. Therefore, this thesis defines semantic similarity through its operationalization in the comparisons of DMs of the SRSs to determine the semantic similarity of these DMs in the case studies presented in this thesis.

The semantic similarity definition and analysis process is discussed in three sections: this section, Section 3.4.1, and Section 5.3. This section provides a definition of semantic similarity of two DMs. Section 3.4.1 describes an operationalization to answer the question of the hypothesis directly, i.e., that the DMs in one set of DMs are more semantically similar to each other than the DMs in the other set of DMs. However this operationalization cannot be understood until more background is given. So, for now a definition of semantic similarity of two DMs is given that

1. is consistent with the later operationalization and

2. provides enough understanding of semantic similarity that allows proceeding to the operationalization.

Accordingly the definition of semantic similarity of two DMs is stated as an *effect* of the operationalization.

Determining the semantic similarity of two DMs in a set of DMs requires some preprocessing on the set of DMs:

1. form the union of all concept names from all the DMs,

2. eliminate syntactic duplicates in the union, and

3. eliminate semantic duplicates in the union and each DM,

to leave lists of unique concepts appearing (1) in the union of all models and (2, ...) in each DM. The determination of semantic duplicates was performed by manually analyzing each concept's

- name,
- attributes,
- methods, and
- relationships with other concepts.

Note, in this work, the term "concept" refers to what is captured inside DMs, i.e., UML OOA class diagrams or conceptual models [62]. Thus a concept can have, among other things, an identifier, i.e., name, methods, attributes, and relationships with other concepts.

The effect of the operationalization is that to evaluate the semantic similarity between two DMs, the two DMs' list of semantically unique concepts are compared in the context of the list of semantically unique concepts appearing in the union of all models. The more concepts the DMs share, the more semantically similar the DMs, and the larger percentage these shared concepts are of the DMs concepts, the more semantically similar are the DMs.

This thesis' operational definition of semantic similarity is fairly close to a generalized definition of semantic similarity [3]:

> "Semantic similarity, variously also called semantic closeness/proximity/nearness, is a concept whereby a set of documents or terms within term lists are assigned a metric based on the likeness of their meaning/semantic content."

The metric used in this work is domain expert opinion combined with manual clustering.

For example, consider a simple domain description:

> There are two baskets in a field. One has 666 oranges in it, and the other one has 1000 oranges in it. I need to know how many oranges there are all together.

Assume that four independent analysts produce four models consisting of concepts:

$d1 = \{Orange, Field, Container\}$

$d2 = \{Orange, Basket\}$

$d3 = \{Orange, Airplane\}$

$d4 = \{Integer\}$

Any pair of DMs from among $d1$, $d2$, and $d3$ are more semantically similar to each other than any pair made of $d4$ and any other DM since $d1$, $d2$, and $d3$ share one common concept. Assuming that for example, both Container and Basket have one common attribute number of oranges, and that the analyst possesses the domain knowledge that a basket is a container, the analyst can assume that a Container is nothing but a Basket. Suppose that the domain expert cannot observe any similarity between Airplane and Basket, or between Airplane and Field. Then DMs $d1$ and $d2$ are more semantically similar to each other than are $d1$ and $d3$ and than are $d2$ and $d3$. Suppose that the domain expert cannot see any link between the Integer concept that belongs to the DM $d4$ and any concepts from DMs $d1$, $d2$, and $d3$. Then, DM $d4$ would not be semantically similar to any of the other DMs.

In summary, this section defines what it means for two DMs to be semantically similar. Section 3.4.1 describes the semantic similarity analysis steps and Section 5.3 provides a formal framework that can be used to replicate the analysis.

1.2 Terminology

Before proceeding any further, it is necessary to establish some vocabulary for the rest of this thesis[2]. The goal of an RE effort is to elicit and analyze requirements,

[2]The terminology used in this thesis is similar to that of the common OOA literature [e.g., 62], and it was reviewed by at least one independent expert [e.g., 69].

and eventually, to specify in a Software Requirements Specification (SRS) document the requirements for the computer-based system (CBS) being built. The portion of the 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 *domain model*, which is a model of the CBS's domain. A UC of a CBS is one particular way some user of the CBS uses the CBS to achieve stakeholders' goals. 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 *statechart* [43], and among the artifacts that are suggested to 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. Conceptual analysis is the activity of discovering and specifying concepts from a domain.

1.3 Why Semantic Similarity of Domain Models is Important

The original idea of OO is that the structure of the software of a CBS models the part of the real world, i.e., the domain, in which the CBS operates, allowing easier validation of the correctness of the CBS and its Software [50, 77, 9, 68, 11]. That is, a flight reservation CBS should have an object for each passenger, flight, aircraft, airport, etc. that is relevant to the functions performed by the flight reservation CBS. As Meyer says, "When software design is understood as operational

modeling, object-oriented design is a natural approach; the world being modeled is made of objects — sensors, devices, airplanes, employees, paychecks, tax returns — and it is appropriate to organize the model around computer representations of these objects." [68, p. 51]

Indeed, it is this relation between a CBS and its domain that is the basis for the advice that the objects and functions of the OO CBS are the nouns and verbs, respectively, of the written description of the CBS and its domain [5, 11]. As Meyer adds soon after the quote above, "... in the physical or abstract reality being modeled, the objects are just there for the picking! The software objects will simply reflect these external objects." [68, p. 51]

Clearly, a DM should be correct and complete. "There are two semantic goals [for DMs]: validity and completeness. Validity means that all statements made by the model are correct and relevant to the problem. ... Completeness means that the model contains all the statements about the domain that are correct and relevant." [65] "The semantic aspect of model quality ensures not only that the diagrams produced are correct, but also that they faithfully represent the underlying reality represented in the domain, ..." [91] If we have several models of a domain, and each is correct, consistent, and faithfully representing reality, we would expect that the models would be semantically similar.

Some researchers talk about a science of modeling [11]. If modeling is to be a science, then we would expect that modeling be reproducible, i.e., that different people would come up with semantically similar models, especially for an objective reality [79] that RE normally assumes [26, 93, 76]. An objective reality is

a reality that is assumed to exist independently of any observer. If modeling is a science then we would expect that models be reproducible, that different people modeling the same objective system will build semantically similar models.

Roughly the same argument can be made from the fact that software engineering is supposed to be a branch of engineering [11]. A goal of any engineering method is to produce predictable results. Predictability for OOA as an engineering method means that given the same domain, different analysts applying the same OOA would be expected to produce semantically similar models of the same domain. Thus, observing the semantic similarity of the results of an OOA method for a domain is a direct measure of the predictability of the OOA method.

As mentioned, a DM should be correct. The question to ask is "How is the correctness of a model to be determined?" The answer is, "by comparison to something that is known to be correct." Is any one person's model to be taken as *the* correct model? Clearly, the notion of correctness of models is one of consensus. A model's is correct if enough people familiar with the modeled domain say that the model is correct. Thus, observing the semantic similarity of independently developed models of the same domain would be one way to validate the correctness of all of the models.

Therefore, semantic similarity seems to be a useful property of a set of independently developed models of a single domain and a fair means of determining the correctness of each member of the set.

Semantic similarity is not new and has been used quite extensively in practice and research. For example, an assumption of semantic similarity of implementa-

tion architectures, which are often derived from DMs, is the basis for the concepts of reference architectures and design patterns [e.g., 34, 17] in the sense that all architectures matching a reference architecture and all design patterns matching a particular design are semantically similar.

In practice, semantic similarity of models is hard to achieve. Grady Booch observes that different observers will classify the same object in different ways [11]. However, these differences in models come mainly from the differences in the perceived purposes of the models. Ostensibly, all domain models built during RE have the same purpose, to model the domain with which the CBS to be built will interact, and to model in a way that allows focus on what is to be built as opposed to how to build it[68]. Therefore, there is reason to expect semantic similarity among domain models of the same CBS's domain.

Finally, Guttorm Sindre, one of the authors of "Understanding Quality in Conceptual Modeling" [65] and of other works dealing with the quality of RE modeling [60, 61], observes in private communication [87], that

- Semantic similarity of models does not prove quality, as the models could be of similarly poor quality. There are many factors other than just pure modeling that affect the similarity and the quality of models. However, it would be reasonable to assume that the fact that two independently produced models are similar at least increases the confidence that both modeling efforts have been performed well and that this confidence grows as the number of independently produced models grows beyond two.

- Semantic dissimilarity of models need not imply poor quality of any of the models, as the models could be for different purposes and from different view points. However, in a situation in which (1) all analysts start from the same and very fixed source of information, and all are asked to produced model with the same and very fixed purpose—as is often done in student exercises or experiments—and in which (2) the semantic quality of the models is measured as the correspondence between the model and the textual description, one might expect a more semantic similarity between independently produced models than in other situations. In this case, it might be possible to argue that semantic similarity of the models increases the likelihood that all are of high quality. In this case, semantic similarity might be seen as a sign of quality not only of the models and modelers but also of the modeling language and modeling method; that is, the models are more semantically similar because the modeling method is clear and easily followed, supporting consistent application of modeling language concepts by independent modelers.

Nevertheless not all agree that semantic similarity is possible or even desirable [e.g., 21, 49, 69]. For such a person, the results of this Ph.D. thesis should be understood as conditional. If one considers semantic similarity of independently produced domain models of a CBS to be a desirable property, this thesis offers a way to increase the chances of achieving it, a way whose effectiveness and costs have been evaluated in an educational setting.

Ultimately the issue is quality, but as Robert Pirsig [78] as quoted by Bhuvan

Unhelkar [91] says, "Quality—you know what it is, yet you don't know what it is. But that is self contradictory.... But some things are better than others, that is, they have more quality... But if you can't say what Quality is, how do you know what it is, or how do you know that it even exists? If no one know what it is, then for all practical purposes, it doesn't exist at all. But for all practical purposes it really does exist.... So round and round you go, spinning mental wheels and nowhere finding any place to get traction. What the hell is Quality? What is it?"

1.4 Motivation

The motivation for this research project comes from the author's observation of students' work on the requirements analysis and specification of a computer-based system (CBS) composed of

1. a telephone exchange or a Voice-over-IP (VOIP) system and
2. its information management system (IMS).

Production of the specification, in the form of a Software Requirements Specification (SRS) document, is the term-long project carried out in the first course of a three-course sequence of software engineering courses that span the last three terms of the undergraduate software engineering program at the University of Waterloo [85]. In later courses, students design, implement, test, and enhance the CBS specified in the SRS.

From 2000 until 2005, the author played a wide variety of roles in this course, serving as customer, group coordinator, UML and SDL instructor, and project evaluator. The author had reviewed over 135 different SRSs, out of over 195 that were developed in this time interval by 3-or-4-student groups of over 740 software engineering, computer science, and electrical and computer engineering students. This educational experience has given him the opportunity to observe various software analysis and specification issues from different perspectives.

The project in the three-course sequence involves using

1. various techniques for developing software for real-time systems and
2. OO techniques for developing information systems.

The real-time components of the telephone CBS are specified using formal finite-state modeling in Specification and Description Language (SDL) [14]. The information-system components of the telephone CBS are specified using the notations of Unified Modeling Language (UML) [83]. Use cases (UCs) [e.g., 62] are used for capturing requirements, and OOA is used as a bridge towards the later OOD. 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 anywhere from 80 to 250 pages.

Through specification reviews, interactions with students, and grading the preliminary and final SRSs, the author has observed many difficulties that arise throughout the specification process. The most frequently observed difficulty is

that of *performing OOA*, i.e., of

1. identifying concepts of the CBS's domain and

2. ascribing the CBS's functionality to these concepts.

This thesis calls this difficulty of identifying concepts and ascribing the functionality, the *concept identification difficulty (CID)*. The result of performing OOA is most frequently captured through the use of DMs. Others have observed similar difficulties [Slide 13 of 70, 48, 54, 59, 6, 32].

1.5 Thesis Overview

This thesis compares and evaluates two typical OOA methods by the degree of semantic similarity of the resulting DMs.

In particular, this work compares the **semantic similarity** of DMs produced from **use cases** by

1. specification of **sequence diagrams** and then DMs[3], and

2. specification of **unified UC statecharts** and then DMs.

Figure 1.1 summarizes the difference between the traditional and new OOA method.

The evaluation is performed on four case studies. The traditional OOA method was carried out for 31 instances of large Voice-over-IP (VoIP) with information management system (IMS) system and for 3 small elevator systems. Data for

[3]In this thesis, OOA method with sequence diagrams is referred to as *traditional* OOA method

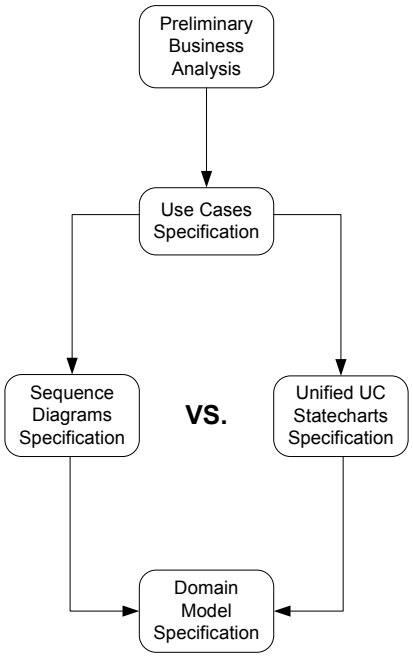


Figure 1.1: Traditional vs. New OOA Method

these were gathered later in case studies CS O31VS and CS O3ES. The new OOA method was carried out for 46 instances of the same large VoIP with IMS system and 12 instances of a medium-sized elevator system. Data for these were gathered later in case studies CS N46VS and CS N12ES.

The main contributions of this thesis are:

- Analysis and interpretation of data that helps us further our understanding and knowledge about OOA methods and the results they produce.
- Evaluation of the impact that traditional functional analysis brings to the OOA and the relationship between them.
- Specification of an OOA method based on the usage of statecharts.

- Analysis of the semantic similarity of independently specified DMs as the means for evaluating predictability of the results of OOA methods used to produce these DMs.

1.6 Thesis Organization

Chapter 2 presents the author's experiences that motivated this work and a discussion of the three practices that helped formulation of the thesis hypothesis. Chapter 3 presents the thesis hypothesis and research method. Chapter 4 describes the unified UC statecharts, a new OOA method, and a case study concerned with the application of the new method to specifying a Turnstile CBS. Chapter 5 presents main case studies and an evaluation of the unified UC statecharts and the new OOA method. Chapter 6 presents background and related work. Chapter 7 presents final conclusions and discusses future work.

Chapter 2

Further Motivation and Preliminary Efforts

Through the observation and evaluation of the groups' work, the author has observed large semantic dissimilarities among DMs produced by the different groups. This thesis calls the inability to achieve a high-degree of semantic similarity of independently specified DMs the *semantic dissimilarity problem (SDP)* of OOA. The author found that the SDP is a result of the CID and is affected by the techniques used to deal with the CID.

The SDP seems to occur during the analysis of all parts of the CBS, independent of the CBS's complexity level. A case study, described in Appendix A and [90], shows that the SDP exists *even in very small CBSs*.

2.1 Discussion of CS O3ES Results

The previously published case study [90] that is described in Appendix A and is named "CS O3ES"[1] suggests that the presence of the SDP in specifications arising from OOA is independent of CBS's size. Even in a small problem such as an elevator CBS, there are many symptoms of the SDP, just as there are in the larger student projects. Certainly, the students who specified VoIP system and its IMS were inexperienced in OOA. However the authors of the SRSs in CS O3ES were writing scientific or pedagogic exemplars of OOA. Therefore, the author found that the *cause* of the CID is neither the size of the specified CBS nor the specifier's lack of OOA experience. Rather, the author observes that the CID arises from two inherent properties of most concepts of complex business CBSs:

1. Each concept fulfills only a sub-activity of a larger activity by interacting with other concepts, e.g., an elevator's motor fulfills only a sub-activity of the overall activity of moving passengers from a floor to another, in collaboration with other elevator's devices, and

2. Each concept participates in many different activities, each for different purposes, e.g., an elevator's door participates in the activity of securing passengers inside the elevator while moving and also in the activity of restricting unauthorized access.

[1]The name of a case study is formed by concatenating: "CS", followed by space, followed by "O" for "Old" or "N" for "New", followed by the number of SRSs evaluated in the case study, followed by an abbreviation of the name of the system analyzed by the case study.

These two properties of the concepts of complex business CBSs contradict what most students learn in their study of OOD and OOP, that a DM should have what is called for the purposes of this discussion "crispness", i.e., well-defined classes, each with a single purpose and a focused set of related responsibilities. Trying to analyze a complex domain with OOA leads many students to discover many of the concepts, responsibilities, and activities that are there but to bend them so that they fit the first decomposition arrived at in an attempt to make a crisp decomposition. In doing so, the typical group of students tends to perform the following actions:

1. The group assigns responsibilities fulfilled through the collaboration of multiple concepts to only one concept. This assignment leads in turn to the following difficulties, each exemplified in the SRSs of CS O3ES:

    - assigning to a concept indirect responsibilities—those achieved by collaboration with other concepts—that are larger in scope than the responsibilities for which the concept is directly responsible, e.g., an elevator cab being responsible for its own movement,
    - missing true responsibilities of that concept; the missing responsibilities are hidden within the larger responsibilities, e.g., an elevator cab being responsible to act as a container for goods and passengers rather than for the control of the overall movement of the elevator cab, and
    - missing other concepts that participate in the overall responsibilities, e.g., missing elevator motor as a crucial component for the overall

activity of moving elevator cab.

2. The group does not capture concepts that are passive, i.e., that are produced or consumed through interaction of other concepts, e.g., not capturing the different time periods required for different timing constraints for the elevator cab movement.

These tendencies appear to be the main cause for the manifestations of the CID and the SDP that the author observed with the typical group's project:

- under-specified analysis models, because the typical group tends to capture only a subset of the available concepts, although many are visible in the requirement artifacts and can be discovered using even the relatively simple noun-underlining technique [62];
- drastically different models of ostensibly the same system from different groups; and
- a large number of software concepts at different abstraction levels.

Notwithstanding that many proponents of OO methods have advocated object orientation as an excellent way of capturing domain concepts and bridging the conceptual gap between computer-based systems (CBSs) and their domains [58, 66, 67], the author's examination of over 135 projects has shown that capturing domain concepts remain difficult even when the practitioner has a good understanding of OO principles and techniques.

It is important to note that these preliminary findings concern OOA only. This thesis makes no claims about OOD, OOP, or any other OO methods. The thesis results arise from an analysis of only the specification artifacts produced as a result of the groups' OOAs of VoIP system and its IMS and from the author's observations of the groups' and students' behavior. There was no opportunity to analyze the designs later produced by these groups. Nevertheless, it appears that none of the students ever had any problems understanding OOP ideas and techniques. However, whenever it came to discovery and analysis of concepts in the domain, it seemed that this OOP background knowledge did not help much. Therefore, the conclusion is that the CID lies neither in the students nor in an apparent inability to understand object orientation, but rather in limitations of the current OOA techniques, at least as applied in RE.

So, if the problem does not lie in analysts' capabilities, but rather in the OOA activity itself and the used techniques, what can we do to reduce the SDP of OOA? This thesis attempts to answer this question and to improve OOA, by offering and evaluating a new OOA method.

2.2 History of the New OOA Method

2.2.1 Early Observations

When carrying out the roles in teaching CS445 from 2000 to 2004, the author had observed in the groups' work that three recommended practices of typical OO methods appear not to work well in practice:

1. iterative analysis,
2. identifying concepts from UCs, and
3. separation of analysis and design.

Iterative Analysis

Occasionally, a group used a typical iterative UC-driven approach to discover domain knowledge and concepts [e.g., 62], while another used a waterfall-like approach [e.g., 82]. The author did not observe any correlation between the number of iterations and the quality of the DMs. One possible explanation is that even when a group uses an iterative approach, and it discovers new artifacts that do not fit into the extant DM, it tends to be hesitant to change the DM. Instead, it tends to try to adapt the new artifacts so that they conform to the existing DM, leading to even more bloated and inconsistent artifacts. Thus, rather than expecting the groups to incrementally improve their DMs over the course of several distinct iterations, the author has learned that it is more effective to encourage them to try to get their DMs right immediately. The author's impression is that when a group knows that it has multiple iterations, it does not put as much effort into getting the DM right the first time, figuring that they will always be able to fix any problems later. However, the problems compound before they get fixed, making them even harder to fix later on. This later problem fixing typically never happens.

Identifying Concepts from UCs

A widely recommended practice that the groups followed is to identify concepts from UCs [e.g., 62]. The purpose of the UCs is to show the interactions between actors and the CBS, while treating the CBS's domain as a black box that does not show internals of the CBS's domain. The UCs in this form capture a subset of the *interchange data* between the CBS's domain and the external actors, which in turn becomes the main source of the concepts. A problem with this approach is that often these concepts are used as if they represent all of the CBS's domain concepts and are used to model the internals of the CBS itself. The *interchange data* do not reflect all of the domain concepts; they reflect only a subset of information that is exchanged between domain and domain's environment. The author has observed that groups cannot rely on only UCs for the discovery of a complete DM. Rather, it is necessary to search for additional sources, and these sources vary widely from project to project. For the groups' SRSs, the additional sources include project-description documents [85] and interviews with the customers. Each of these additional sources is usually less structured and less consistent than UCs, making it even harder to verify the consistency and completeness of the DM. Thus, UCs are quite limited in scope, and deriving them should be only one of many steps taken to find or generate artifacts that can help in conceptual analysis.

Separation of Analysis and Design

The impact of choosing the perspective from which analysts define the boundary between CBS and its domain was present in each group's work. The typical group takes one of two evident perspectives. The first perspective treats the CBS and its domain as one system together. A follower of this perspective tends to assume that a domain concept, i.e., an analysis concept, is a CBS's concept, i.e., a design concept, and vice versa. The author observed that taking this perspective usually results in a reduced ability to distinguish between analysis and design concepts; compared to the second perspective, a smaller set of analysis concepts is captured in the DM.

The second perspective treats the CBS and its domain as two different systems. A follower of this perspective tends to consider analysis concepts and the design concepts to be different kinds of concepts, and thus tends to capture more analysis concepts in the DM than a follower of the first perspective.

The capture of more analysis concepts that results from following the second perspective suggests that it is helpful to have a clear separation between a CBS and its domain, i.e., to have a clear separation between the analysis and design concepts. Thinking of a CBS as a direct simulation of its domain, i.e., that the CBS and its domain are the same systems, results in a lower quality DM with fewer discovered analysis concepts. Although they may appear highly similar, a CBS and its underlying domain are different spaces; analysis concepts and design concepts are not only different concepts, they are different *kinds* of concepts.

2.2.2 The Origins of a New OOA Method

When working with some of the groups as a project teaching assistant (TA), the author suggested a method that helped them increase the semantic similarity and completeness of their DMs:

1. Clearly separate the domain from the CBS;
2. describe and explain domain processes in addition to the UCs;
3. describe the CBS in terms of logical software processes based on the domain processes;
4. discover mappings between the domain processes and the CBS processes; and then
5. break each of the domain and the CBS into concepts and structural components.

That is, process analysis is performed before performing conceptual analysis for both the domain and the CBS. This change of method helps an analyst move away from the discrete superficial breakdown of the domain into actors and their activities captured through UCs towards what the analyst actually has to analyze. The analyst should understand two distinct systems:

1. the domain, and
2. the CBS.

Completing process-based analysis of both CBS and its domain before attempting the conceptual analysis of the domain is probably too radical a change, taking into account that a standard approach is to perform conceptual analysis immediately after UC specification. However, the author realized that the core idea of attempting to derive some intermediate analysis artifacts that are more constrained than DMs might be worth exploring.

The main problem is finding exactly which modeling paradigm and what artifacts should be produced before pursuing with OOA. The paradigm should

1. minimize the amount of the intermediate work,
2. be a commonly used analysis paradigm within the OO paradigm, and
3. provide a mechanism for a smooth transition to conceptual analysis.

There are several kinds of modeling that might satisfy these requirements:

1. architecture-driven modeling,
2. procedural modeling,
3. state-based modeling, and
4. goal-driven modeling.

Each of these modeling paradigms might be used as an intermediate tool for constraining conceptual analysis and OOA in general. Perhaps this method of using different modeling paradigms rather than the same paradigm for moving from domain requirements to DMs will result in higher quality and more semantically

similar DMs of the same domain. The same way of thinking might be beneficial also for production of other types of development artifacts. For example, using three different paradigms for analysis, design, and implementation might result in more consistent, complete, and semantically similar artifacts at each stage than what can be achieved using only the OO paradigm for all three.

2.2.3 Observations about the New OOA Method

The 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 often results in a SRS that is difficult for both customers 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., 22] 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 [e.g., 22].

Extending this reasoning, the author realized that maybe some *new artifact* based on the UCs would allow analysts to produce *even* more complete, consistent, and correct SRSs, and DMs 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

- 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 concepts, and
- find the *big picture* behavioral domain model.

In the typical UC-driven requirements analysis method, UC discovery is followed by drawing sequence diagrams for the UCs and breaking down the domain's side of the UCs into the domain's components, to yield a DM of the CBS [e.g., 62]. This kind of approach had been taught to the students for several years. A less common alternative method is to follow UC discovery by drawing UC statecharts [28, 62, 41]. Nevertheless, in each method, a UC is an artifact at the widest *scope*. *Scope* refers to the number of functional requirements captured and specified using an artifact. A sequence diagram, a 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 the UC. To arrive at the big picture behavioral domain model, it was necessary to proceed in the opposite direction. Rather than decomposing the UCs, as suggested in

many UC-driven requirements analysis methods, the author realized that it might be better to unify the UCs into a behavioral domain model using statecharts, as suggested in other methods [39, 97, 47].

So, the author decided to introduce a method based on performing detailed behavioral analysis of domain through the unification of the behavior described in UCs into an integrated behavioral domain model using statecharts. Once the author decided to use statecharts as the notation in which to unify the UCs, the author had to develop an unification method, to apply it in practice, and to evaluate the results. The method, which builds on using statecharts to model UCs and then unifying the UC statecharts into a unified UC statechart [39, 97, 47], is called "UCUM (Use Case Unification Method)".[2]

UCUM is derived from several sources. UCUM is primarily Larman's UC-driven iterative method [62]. The principles of constructing a unified UC statechart are based on Douglass's and Gomaa's principles of UC statechart construction [28, 41]. The underlying behavioral domain model semantics is based on Glinz's [38, 40]. The statechart syntax and semantics follows UML 2.0 standard [83].

The idea of unifying UCs into a unified UC statechart is due to Glinz; Whittle and Schumann; and Harel, Kugler, and Pnueli [39, 97, 47]. The students did not follow any of these authors' proposed methods in particular. Instead, the students used the common ideas of all these methods to unify UCs into a unified UC stat-

[2]The method is named only to make it easier to distinguish it from the other methods mentioned in this thesis.

echart. Rather than having students follow a formal method, the idea was to have students tackle building unified UC statecharts as an engineering problem that they had to solve. The author used a small domain as an exercise during the tutorials to have students work from scratch on applying and refining a simple and practical method for unifying use cases into a unified UC statechart. This exercise ended up being the first case study described in Chapter 4, which introduces UCUM and provides results of this initial exercise of specifying a behavioral domain model using statecharts.

So, UCUM was presented and refined during three tutorial sessions with almost 150 students in attendance over the three sessions. UCUM was tailored through the 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 in her work on her main course project. Only after specifying UCUM and only after completing all case studies, the author compared the students' results with those of the sources of UCUM. This comparison is presented as part of the description of related work in Chapter 6.

Chapter 3

Research Method

The research method is an after-the-fact analysis of the work done by students before and during an attempt by course instructors and the author to improve the students internalization and performance of OOA in a course teaching OOA as a RE method.

Section 3.1 reiterates the problem summary as defined in the Chapter 2. Section 3.2 poses the research question and Section 3.3 states the thesis hypothesis to be validated by the research. Section 3.4 outlines the after-the-fact case studies performed to validate the hypothesis, research steps, and provides this thesis' definition of semantic similarity. Finally, Section 3.5 analyzes the threats to accepting the conclusions of the case studies.

3.1 Problem Summary

As discussed in Chapter 2, the most frequently observed difficulty is that of *performing OOA*, i.e., of

1. identifying concepts of the CBS's domain and
2. ascribing the CBS's functionality to these concepts.

This thesis calls this difficulty of identifying concepts and ascribing the functionality, the *concept identification difficulty (CID)*. The result of performing OOA is most frequently captured through the use of DMs.

Through the observation and evaluation of the groups' work, the author has observed large semantic dissimilarities among DMs produced by the different groups. This thesis calls the inability to achieve a high-degree of semantic similarity of independently specified DMs the *semantic dissimilarity problem (SDP)* of OOA. The author observed that the SDP is a result of the CID and is affected by the techniques used to deal with the CID.

The SDP seems to occur during the analysis of all parts of the CBS, independent of the CBS's complexity level. A case study, described in Appendix A and [90], shows that the SDP exists *even in very small CBSs*.

3.2 Research Question

The thesis work asks:

How can we reduce the SDP? That is, how can we achieve more semantically similar DMs?

Why is this an important question? One can argue that there are three important roles of a DM. The first role is to help an analyst understand a domain. The second role of a DM is to help communicate the important domain concepts among the developers, i.e., to be used to share domain knowledge. The third role of a DM is to help move from the domain's requirements to CBS's design artifacts. Each role is tightly linked to our understanding of the domain as captured through the gathered requirements.

An impediment to communication of analysis models is the difference, i.e., the lack of semantic similarity, among even mental analysis models created by different analysts. Sharing understanding of the system through analysis models is tightly linked to the analysis models' use for design purposes, and semantically dissimilar analysis models can yield many different design models. As such, it is useful to have the means that lead all analysts to a semantically similar analysis model as early as possible; ideally, this convergence should occur even before they start any communication.

Why is the DM so important? OO methods typically advocate mapping the DM directly to the software design class and object models. Each of these models, in turn, drives the generation of other OOA and OOD models. This strong influence causes any errors and misunderstandings captured in the DMs to be propagated to all other OOA and OOD models. This propagation ultimately negatively affects the CBS's implementation.

3.3 Thesis Hypothesis

The thesis hypothesis is:

> **The SDP can be reduced by performing a detailed behavioral analysis before conceptual analysis.**

The contribution of this thesis work is the analysis of the benefits, drawbacks, and side effects of modifying the traditional OOA process by performing detailed behavioral analysis before conceptual analysis.

3.4 The Research Method

This thesis research was carried out in two phases. The first phase is the development of UCUM carried out in an effort to improve the teaching in the CS445 course, and the second phase is the evaluation of UCUM in an after-the-fact analysis of the work by students before and after the introduction of UCUM. This analysis of the students' work was manual and qualitative because an automated, quantitative analysis would require taking some aspects of the CBSs specified out of context. Also, until the analysis has been done many times, it is not even clear what of the analysis can be automated. The analysis was carried out in four case studies of previously written SRSs:

1. **Preliminary semantic similarity analysis phase and small elevator systems case study** — This case study involves analysis of 3 different specifi-

cations of 3 different small elevator CBSs. This previously published [90] case study is described in Appendix A and is named "CS O3ES".

2. **UCUM's specification phase and turnstile system case study** — This case study involves three specifications of the Turnstile CBS [89], produced collaboratively by the author and the students attending tutorial sessions of the CS445 and the CS846 classes. This case study is named "CS N3TS".

3. **The individual elevator system case study** — This case study involves 12 medium-sized specifications of the controller for a two-elevator CBS in a low-rise building, produced as individual term-long projects in the CS846 class. This case study is named "CS N12ES".

4. **The group VoIP system case study** — This case study involves 46 large-sized specifications of a VoIP system and its IMS, produced as group term-long projects in several offerings of CS445 class. This case study is named "CS N46VS". The DMs in the SRSs of CS N46VS are compared to the 31 DMs of the same system specified using traditional OOA.

3.4.1 Semantic Similarity Analysis

Answering the research question and testing the hypothesis requires comparing the semantic similarities among the DMs in one set of DMs, produced by OOA *without* UCUM, with the semantic similarities among the DMs in another set of DMs, produced by OOA *with* UCUM to see in which set are the DMs more semantically similar to each other. Rather than computing the semantic similarity of

each DM in the two sets of DMs, rather than computing some aggregate semantic similarity among the DMs in each set of DMs, the analysis attempts to directly determine in which set are the member DMs more semantically similar to each other, and then to estimate by how much.

This analysis requires some preprocessing on *each* set of DMs:

1. form the union of all concept names from all the DMs,
2. eliminate syntactic duplicates in the union, and
3. eliminate semantic duplicates, i.e., all names that represent the same concept in the union and each DM are uniformly replaced in each DM by one of the names, that is chosen as the representative name for the concept.

to leave lists of semantically unique concepts appearing (1) in the union of all models in the set and (2, ...) in each DM in the set. The determination of semantic duplicates is performed by manually analyzing each concept's

- name,
- attributes,
- methods, and
- relationships with other concepts.

To decide in which set of DMs are the elements more semantically similar to each other, a variety of measures are computed on the concepts that the two preprocessed sets have in common. Each measure captures the intuition that

- the more concepts the DMs in a set share, the more semantically similar to each other are the DMs in the set, and
- the larger the percentage these shared concepts are of all the concepts in the DMs, the more semantically similar to each other are the DMs in the set.

A more detailed description of this analysis is found in Section 5.3.

3.5 Threats to Validity

The research of this thesis is a retrospective, after-the-fact, analysis of work performed during courses for which the author was a TA. Clearly, 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. In particular, the course

1. did not require the use of strict method presented in tutorials upon the students, beyond the basic core UCUM,
2. did not require any particular group organization or division of work,
3. did not limit the size of the CBS and its domain being specified.

Finally, each analyzed DM 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 this case study 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.

Chapter 4

Domain System Statecharts

In many a UC-driven requirements analysis method [e.g., 62], the first task an analyst performs in modeling the behavior of the CBS being built is to write UCs that describe the CBS's intended behavior. From these UCs, the analyst begins to model the entire CBS with the goal of writing a SRS describing the CBS. One of the specification artifacts is the DM of the CBS's domain.

A UC describes a use of a CBS from at least one user's perspective. Domain experts and analysts together typically capture UCs during and after requirements elicitation from many stakeholders, each with a different perspective. The author has 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 due to 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., 64]. 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 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. Also, the use of UCs in the preliminary case study and their widespread use in practice has led to the constraining of UCUM to be based on UCs too.

UCUM is based on a very 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 described with a UC statechart [e.g., 28, 41], then it should be possible to unify these UC statecharts into a unified UC statechart that describes a high quality behavioral domain model[1] [39, 97, 47]. 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, the author found many a student skipping the production of UC statecharts and still producing a good unified UC statechart. Douglass [28] summarizes the advantages of specifying a UC's behavior using a UC statechart in a single paragraph:

[1]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 statecharts widens rather than narrows scope, in terms of the number of functional requirements taken into consideration. Widening the scope leads to exposing problems that might still exist in the individual UCs in the same way that widening the 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.

This integration method and the increased complexity of the resulting model, due to the number of functional requirements and their inter-relationships, helps detecting missing requirements and discovering inconsistencies.

The rest of this chapter discusses the semantics of unified UC statecharts and then describes the process of UCUM and describes the case study, CS N3TS. Note that the author defined the semantics of unified UC statecharts only *after* the case

studies were finished. The author *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 not sufficient even for the specification of the unified UC statecharts in the SRSs of CS N3TS due to the amount of 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 [44], 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. 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." [28] In practice, states are used to capture, for example, any configuration of the object's variables or any activity occurring within the system [e.g., 28].

For the purposes of behavioral domain modeling, the most appropriate semantics for unified UC statecharts can be described in terms of *postconditions* as a particular kind of goals.

Goal-driven RE [e.g., 72, 94] 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 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 method focuses on determining CBS requirements through analysis of users' *personal* and *business* goals, the goal-driven RE has been used to enhance traditional RE methods, among which are UC-driven requirements analysis methods [e.g., 22]. 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 part of the unified UC statechart followed.

Postconditions capture the *intention* and the *target condition* for the entity under analysis. For example, in the case of an elevator system, a postcondition for an elevator is to deliver passengers to their requested floors. This postcondition captures both the *intention* of delivering passengers and the *target condition* of arriving at the passengers' requested floors. This particular postcondition 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 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 unified UC statechart has on the CBS. While the semantics of unified UC statechart nodes is different from that in traditional statechart semantics, the semantics of unified UC statechart transitions is consistent with that in traditional statechart semantics; and, in particular, in all presented case studies, statecharts conformed to UML 1.x or UML 2.0 statechart syntax and semantics.

"Why not use UML activity diagrams [83] 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 inter-

pretations; e.g., an activity diagram can be viewed as a statechart, as a Petri net, or as a flowchart [28].

- Laying out and managing a large activity diagram is more difficult, in the author's 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 [28]. 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 statecharts.

Moreover, the author did not find any features provided by activity diagrams that are not provided by statecharts.

4.2 Process

UCUM emerged from the author's literature review, practice on the author's own examples, and the preliminary work with students in CS N3TS. The author 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 is for only the exposition here. 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.

4.2.1 Process Steps

For the CBS *S* to be built:[2]

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 subsystem.

- Group the UCs into domain subsystems according to the UCs' business concerns. This grouping yields the first level of the decomposition of *S*'s

[2]In the list below, the text following "Step *n*:" is a summary of the process described in the bulleted list headed by "Step *n*".

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* [62] 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 steps, redundant steps, inconsistent terminology and improper ordering of steps.

- Simplify the unified UC statechart using concurrent and sub-machine states.

To reduce unified UC statechart 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.

Step 5: Perform conceptual analysis of the unified UC statechart.

- Map domain subsystem from the UC diagram to the unified UC statechart.
- Perform conceptual analysis of the unified UC statechart.

The goal of the Step 5 is to assign each activity in the unified UC statechart to a concept and to capture each data appearing in the unified UC statechart as concept. The main and most important difference between UCUM and traditional OOA is that conceptual analysis is performed on unified UC statechart rather than on UCs as in traditional OOA.

4.3 Turnstile Case Study: CS N3TS

The case study CS N3TS is about the collaborative production of 3 unified UC statecharts and 3 DMs during 3 tutorial sessions of the CS445 and CS846 courses[3]. The 3 unified UC statecharts and DMs were of the same CBS, the Turnstile CBS,

[3]Note, the process description includes all steps as recommended to the students for their projects. 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.

described in an example SRS at the course Website [4]. The starting point for building unified UC statecharts was the set of UCs identified in the example SRS. The first unified UC statechart and DM were produced by 12 CS846 students, the second unified UC statechart and DM were produced by about 80 CS445 students, and the third unified UC statechart and DM were produced by about 50 CS445 students.

The primary value of the exercise was observing:

- three different groups of about 150 students altogether thoroughly analyzing a small CBS's domain to produce unified UC statecharts and DMs, and
- the feedback the production of these unified UC statecharts had on the UCs in the earlier, Website-published, and supposedly polished UCs [4], and the impact of the conceptual analysis of unified UC statecharts on the resulting DM.

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 and a DM. While the author tried his best to let the students go where they wanted, the author did step in to prevent them from going too far astray, and the author did ask questions that helped them notice things that the author could see they were overlooking. The case study, CS N3TS, is a look back at what happened for the purpose of arriving at the description of UCUM given in this thesis. This case study does not include the analysis of the semantic similarity because the facilitated way the unified UC statecharts and DMs were produced

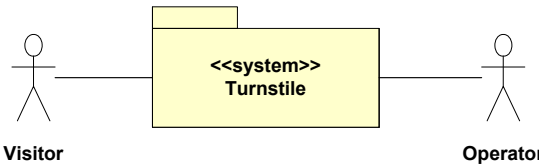
practically guarantees their semantic similarity. However, it does compare the UCUM produced DM to that in the published SRS whose UCs were the starting point of the exercise.

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 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.

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

4.3.1 Step 1

The first step is to **specify UCs**. Figure 4.1 shows the three main UCs from the original Turnstile CBS SRS [4] and the context diagram with 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. The Turnstile CBS consists of both Turnstile hardware and control software. The Turnstile hardware consists of three units, the Paybox, Housing, and Barrier.



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 4.1: Use Cases

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

Figure 4.2 shows the main UC diagram with the CBS's domain boundary defined. During the tutorials, students attempted to group UCs according to their business concerns, but they were not able to agree on how to do it. Therefore, they decided to proceed on without grouping UCs, with the understanding that

the grouping could be done later if it proved necessary or helpful. The only proposed grouping, suggested during one of the tutorials, was to group 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 for 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 CBS of CS N3TS played a role in the difficulty of grouping 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 CBS was perfectly clear without the help of the domain subsystems.

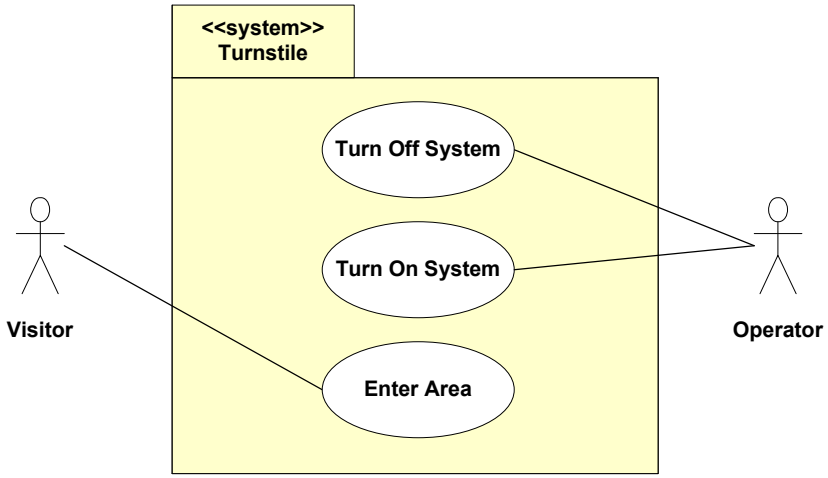


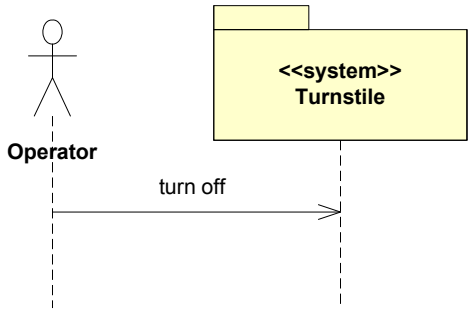
Figure 4.2: Use-Case Diagram

4.3.3 Step 3

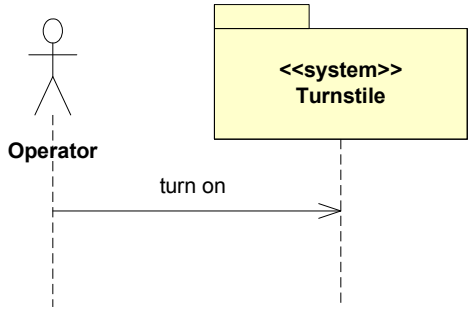
The third step is to **define UC system sequence diagrams**. The goal of this step is to define the domain's external interface. It is important to clearly identify *input* and *output* events for the specification of the unified UC statechart in the next step.

Figure 4.3 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 steps 2 and 3 of UC1, and steps 3, 4, 5, 6, 7 and 8 of UC3. These 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.

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

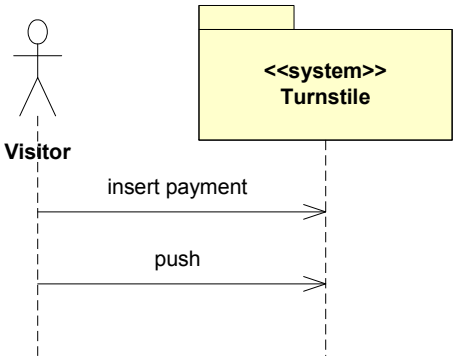


Figure 4.3: System Sequence Diagrams

4.3.4 Step 4

The fourth step is to **specify the unified UC statechart**. This was the main and most difficult step. For each UC, the students had to merge UC with the unified

UC statechart built so far. Optionally, the students may have chosen to explicitly specify the UC statechart before merging it with unified UC statechart.

If the students detected any problems with UCs, system sequence diagrams, or system boundary definition, they were supposed to fix them. Most often these are problems that can be found in the UC being currently merged with unified UC statechart — problems such as inconsistent abstraction levels, missing steps, redundant steps, improper step order, and any other problems detected while attempting to merge UC activities with the unified UC statechart. After each UC was merged, the students were suggested to attempt to simplify unified UC statechart using concurrent and sub-machine states.

In addition, the students were recommended to:

- 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 sub-machine state. The recommendation was to minimize required rework if some activity needs further decomposition.

- 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 sub-machine state. The recommendation was to minimize required rework if some activity needs further decomposition.

Figures 4.4 through 4.23 show the step-by-step construction of the unified UC statechart for the Turnstile CBS. The description of each step discusses all choices

for the step and the impact of the step on other artifacts.

Figures 4.4 and 4.5 show 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 the number of students wanting to start with UC2 was larger. 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 4.4 shows the system sequence diagram for the Turn On System with the identified turn on external event that initiates the UC. Figure 4.5 shows the turn on event as the first step of UC2 and as the initial event in the partial unified UC statechart built so far. Figure 4.5 shows also capturing the second step of UC2 as the accepting events state.

The second step 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 is accepting external events, and
- the term "again" was indicating tight coupling with some other UC.

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

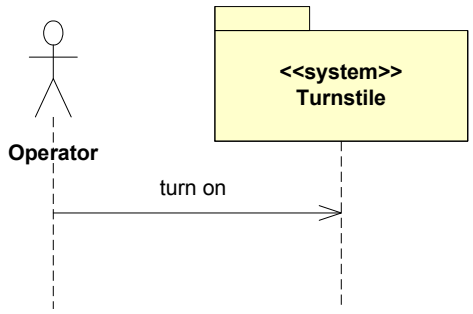


Figure 4.4: UC2 System Sequence Diagram

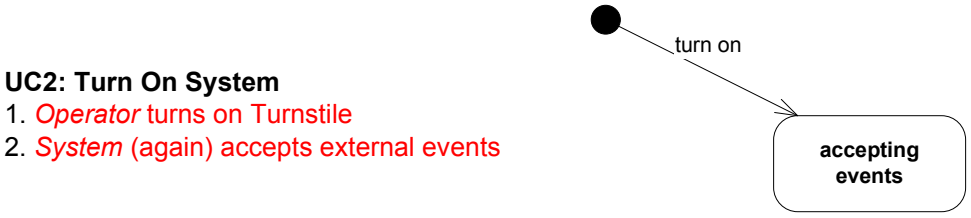


Figure 4.5: Building Unified UC Statechart (1)

The next UC that students decided to tackle is UC3. Figure 4.6 shows the system sequence diagram for UC3 with two discovered external events, insert payment and push.

Figure 4.7 shows how the first external event, insert payment, was integrated into the partial unified UC statechart. The students observed that the state accepting events of UC2 does not capture the intent of the event of the Step 2 of UC3 and cannot be merged to the unified UC statechart due to a difference of abstraction levels and concerns, i.e., the first part of the Step 2 is a general observation about accepting events while the second part of the Step 2 captures the domain-specific functionality of processing a payment.

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

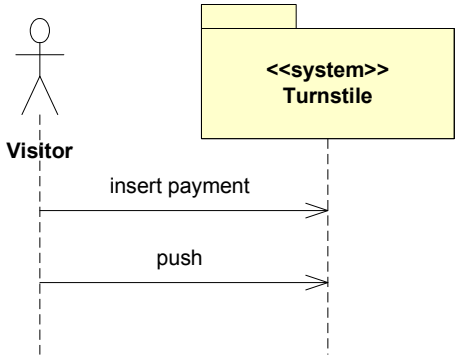


Figure 4.6: UC3 System Sequence Diagram

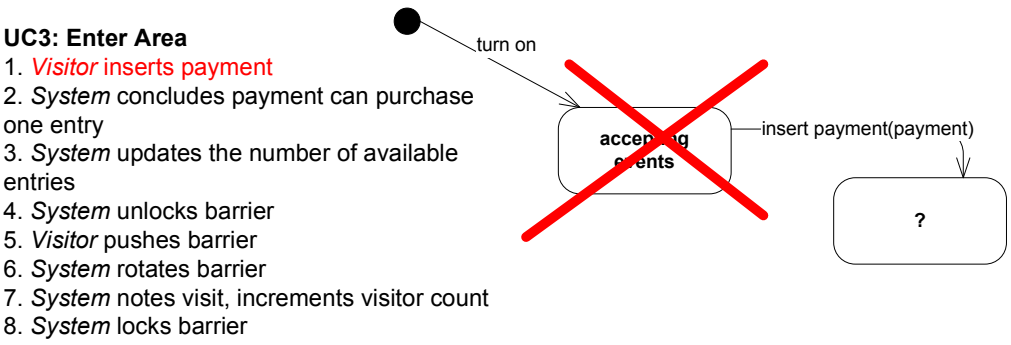


Figure 4.7: Building Unified UC Statechart (2)

Figure 4.8 shows the modifications of Step 2 of both UCs. Figure 4.8 shows also the modification to Step 3 of UC3. The students realized that Step 3 of UC3

was written at the same abstraction level as Step 2 and a part of the same activity. Therefore, Step 3 of UC3 should be merged with the new Step 2. The students captured the next activity of processing payment as the new Step 2 of UC3, thus unifying the old Steps 2 and 3 of UC3. The new processing payment activity was captured as a sub-machine state. The sub-machine state was expected to be decomposed later and was expected to include the old Steps 2 and 3 among other activities in the decomposition.

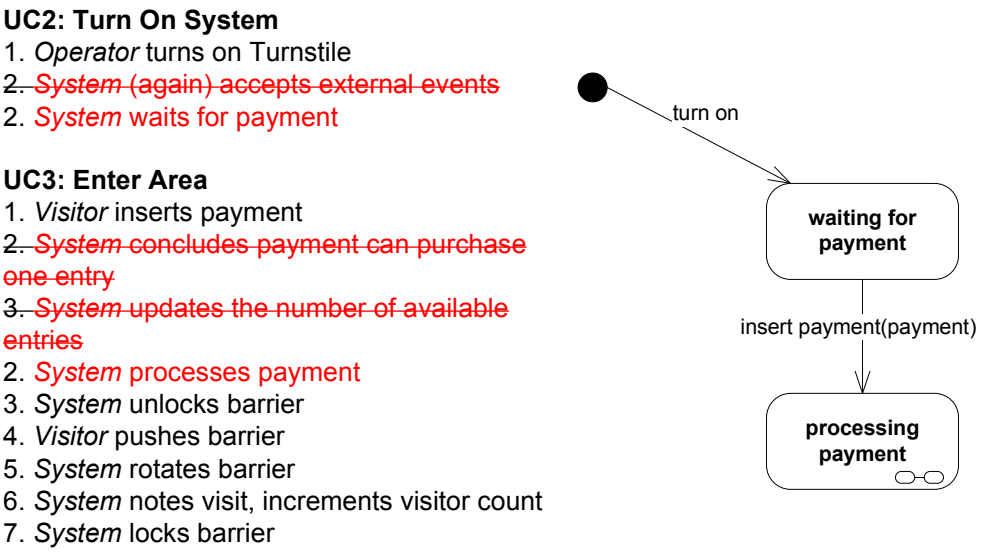


Figure 4.8: Building Unified UC Statechart (3)

Figure 4.9 shows the step of refining unified UC statechart using a composite state. The students judged the previous modification to Step 2 of UC2 UC to be at an abstraction level lower than the originally intended for UC2 and of a different business concern. Therefore, the students decided to introduce a new higher level state controlling access and to modify UC2 and 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 Step 2 of UC2.

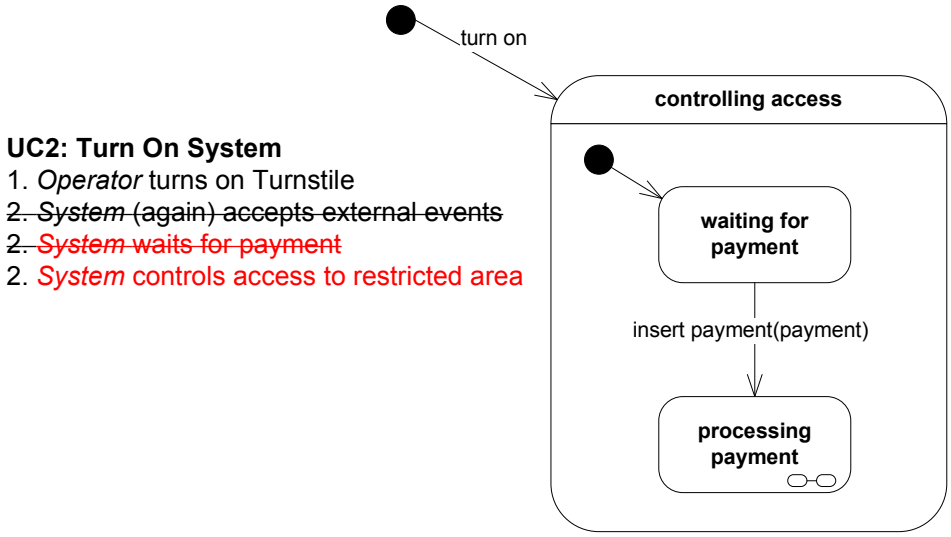


Figure 4.9: Building Unified UC Statechart (4)

Figure 4.10 shows how the students proceeded with the integration of Step 3 of UC3 into the unified UC statechart. The statement unlocks barrier immediately after processing payment was recognized as a big logical gap within the context of the state machine. What was missing was the conclusion of a successful payment that deserves unlocking of the barrier and the notification of Visitor of the barrier being unlocked. Modification was made to Step 3 of UC3. The students also updated the system sequence diagram (SSD) for UC3 to show notification to Visitor, as shown in Figure 4.11.

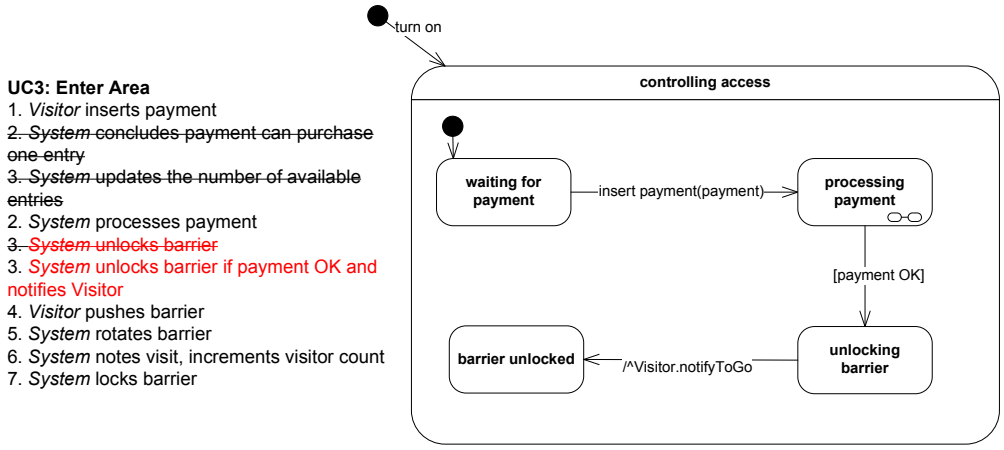


Figure 4.10: Building Unified UC Statechart (5)

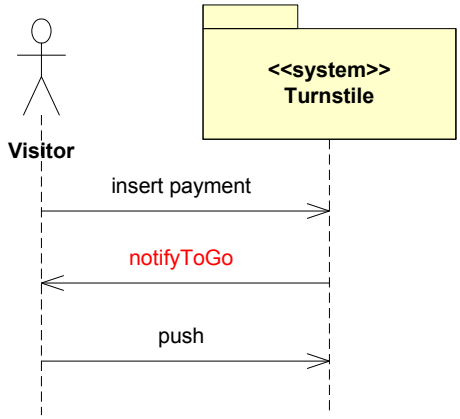


Figure 4.11: UC3 System Sequence Diagram

Figure 4.12 shows the refinement of Step 3 of UC3. Dealing with the successful payment option raised the issue of dealing with an alternative when payment is not sufficient and money should be returned. The change was incorporated into Step 3 of UC3. Students initially captured this alternative as an activity shown

with transition and state drawn in red[4] color in Figure 4.12. Since this unified UC statechart specification was done as primarily 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*.

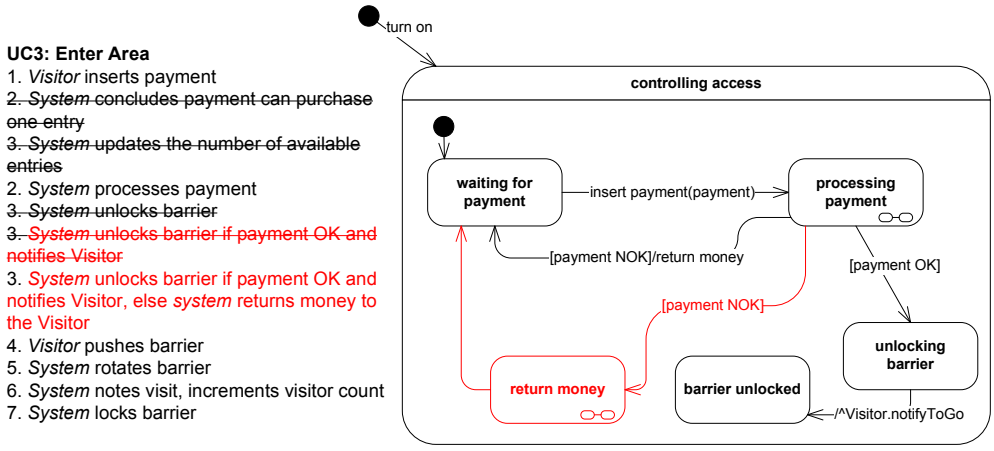


Figure 4.12: Building Unified UC Statechart (6)

Figures 4.13 and 4.14 show the modifications to Steps 4 and 5 of UC3 and their integration into the unified UC statechart. At first, it appeared that Steps 4 and 5 could be integrated in a straightforward fashion as depicted in Figure 4.13, but analysis of both UC3 and the unified UC statechart pointed out the logical problem of the System instead of Visitor rotating the Barrier. The students noticed that the Barrier is an external entity. Therefore, it should be outside of the system boundary. The students modified Steps 4 and 5 as depicted in Figure 4.14.

[4]If you are looking at a black and white copy, what is described as red appears as gray.

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

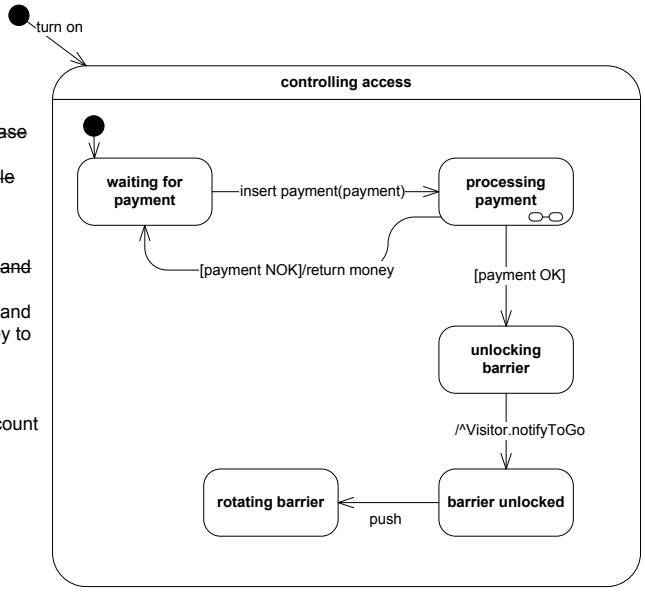


Figure 4.13: Building Unified UC Statechart (7)

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

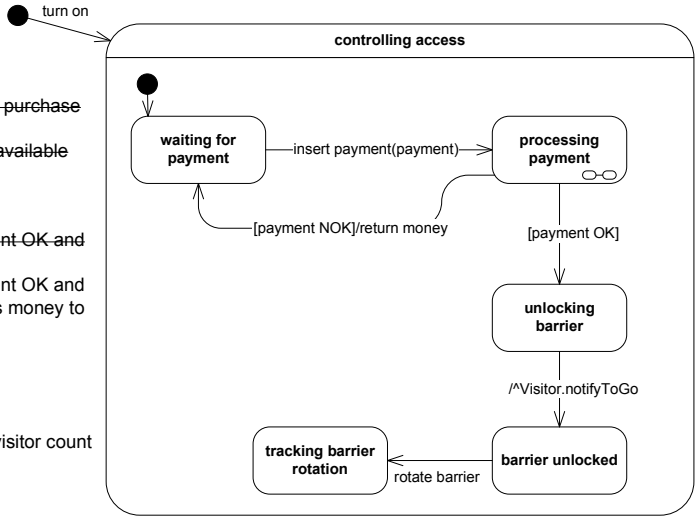


Figure 4.14: Building Unified UC Statechart (8)

Since the analysis of Steps 4 and 5 and their integration into unified UC stat-

echart raised the issue of Barrier being external to the system's boundary, the students decided that they should exclude Barrier from the system boundary definition. The students updated the context diagram as shown in Figure 4.15, and system sequence diagram for UC2 and UC3 as shown in Figure 4.16.

Now, since the students had excluded Barrier from the system boundary definition, Barrier became an *actor*. After some discussion, the students realized that although they had wrong actor for this particular UC, i.e., Visitor instead of missing Barrier, the change could be ignored at the level of the UC and the rest of the unified UC statechart was deemed correct. The reason for this ignoring was that the only responsibility of Barrier was to act as a user interface without providing additional complex functionality. The only required change was to indicate additional system outputs to new actors as shown in Figure 4.16.

So, the students defined a new system boundary definition, in which the CBS under specification is Turnstile Controller. They realized that the old system definition was in fact a *business system* that included Barrier, Paybox, and Switch, as shown in Figure 4.15. Again, the students decided that there was no need to change the UCs since the *new actors*, Barrier, Paybox, and Switch, merely acted as *user interfaces* between the *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 what the actors and system boundary really are. This movement is not surprising, as for example, we can observe the same tendency of ignoring 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.

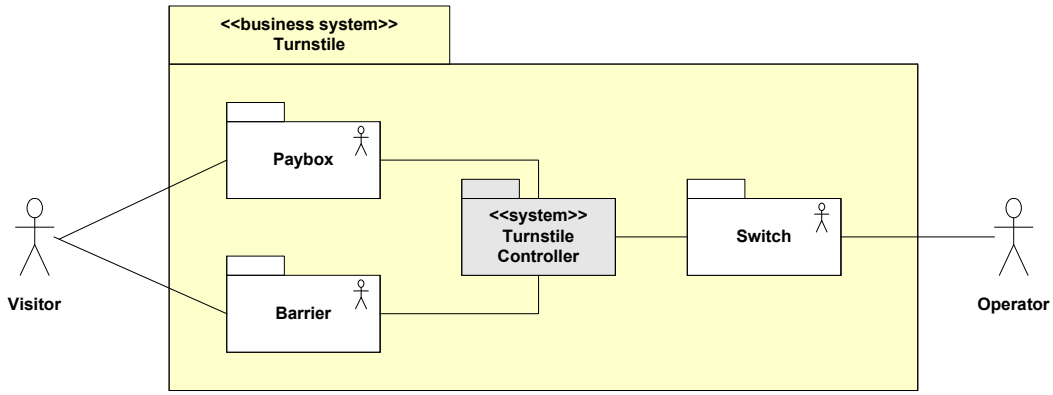


Figure 4.15: Redefined System Boundary

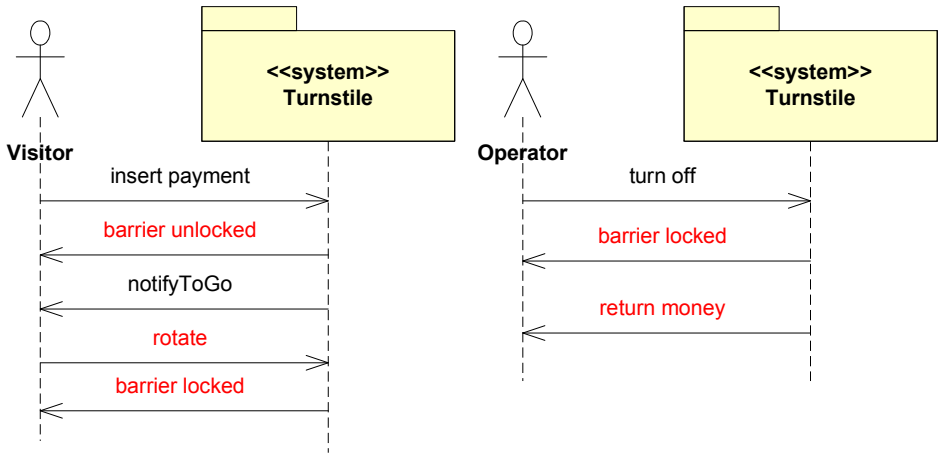


Figure 4.16: UC2 and UC3 System Sequence Diagrams

Figure 4.17 shows the simplification of the second part of Step 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 step, noting visit activity. Some students judged

incrementing visitors count as *a part of* the noting visit activity. Therefore, noting visit activity needed to be decomposed further during the later refinement of the unified UC statechart.

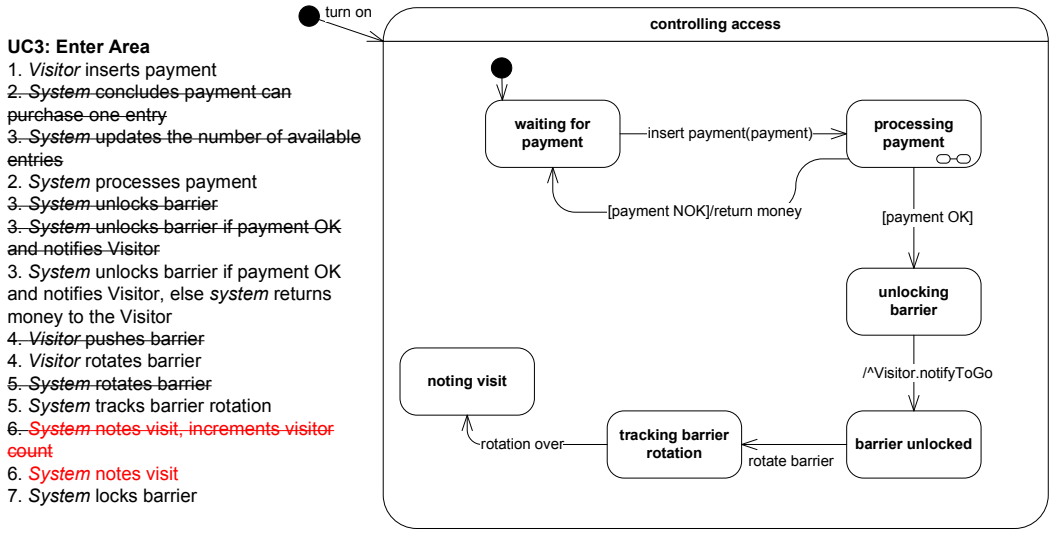


Figure 4.17: Building Unified UC Statechart (9)

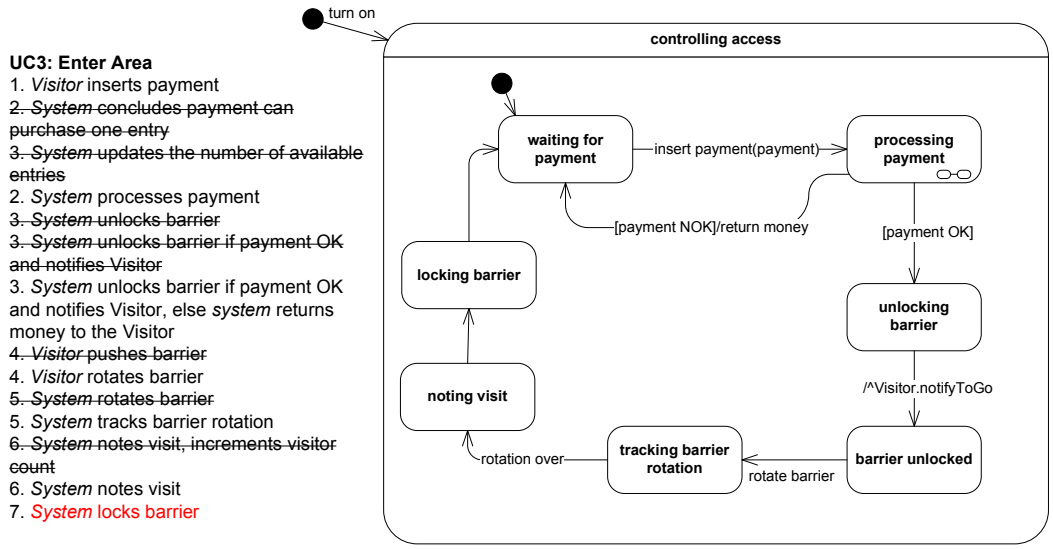


Figure 4.18: Building Unified UC Statechart (10)

At this point, the students were ready to tackle the last UC, UC1. Figure 4.19 shows integration of Step 1 of UC1 into the unified UC statechart. Deciding from which state to send the turn off event indicated that the earlier decision of treating UC2 to be at a higher abstraction level than UC3 and of introducing the composite controlling access state was very useful. The introduction of the composite state would have been required at this stage any way since turn off event has to be handled from any state in 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 Step 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 step to a higher abstraction level and to postpone its decomposition into details. They captured Step 2 as the resetting activity, which needed further decomposition.

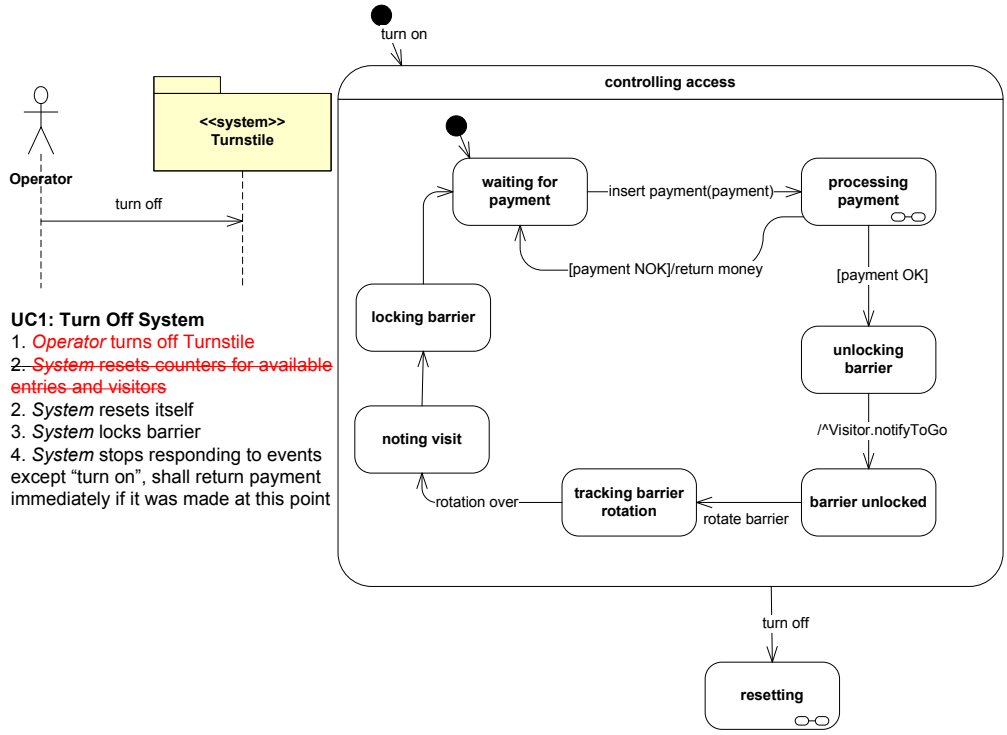


Figure 4.19: Building Unified UC Statechart (11)

Figure 4.20 shows that the students judged also Step 3 to be a part of the resetting activity, and the Step 4 became the new Step 3 in the modified UC1.

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

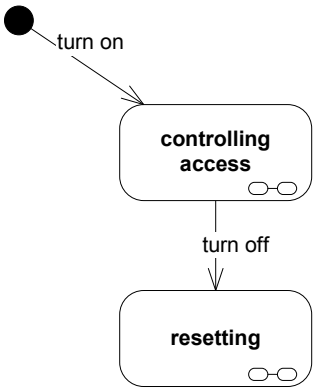


Figure 4.20: Building Unified UC Statechart (12)

Figure 4.21 shows that the first part of the new Step 3 introduced the need for a new state off, while the second part of the new Step 3 was judged as redundant and possibly a part of the previous resetting activity that was to be decomposed later.

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

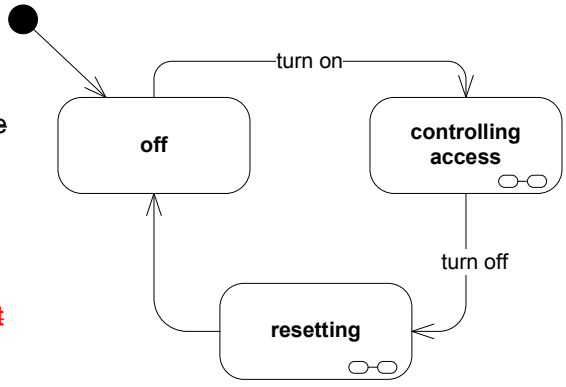


Figure 4.21: Building Unified UC Statechart (13)

Finally, the resetting activity in the UC1 exposed the need for the introduction of a corresponding setting up activity in UC2, as shown in Figure 4.22.

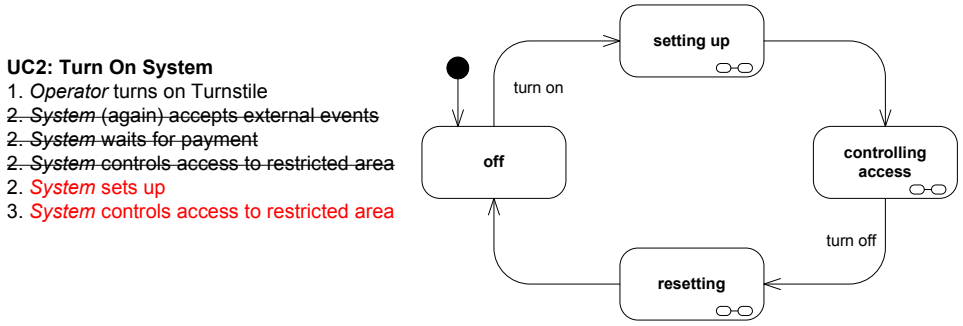


Figure 4.22: Building Unified UC Statechart (14)

Figure 4.23 shows the final, integrated unified UC statechart of all three UCs.

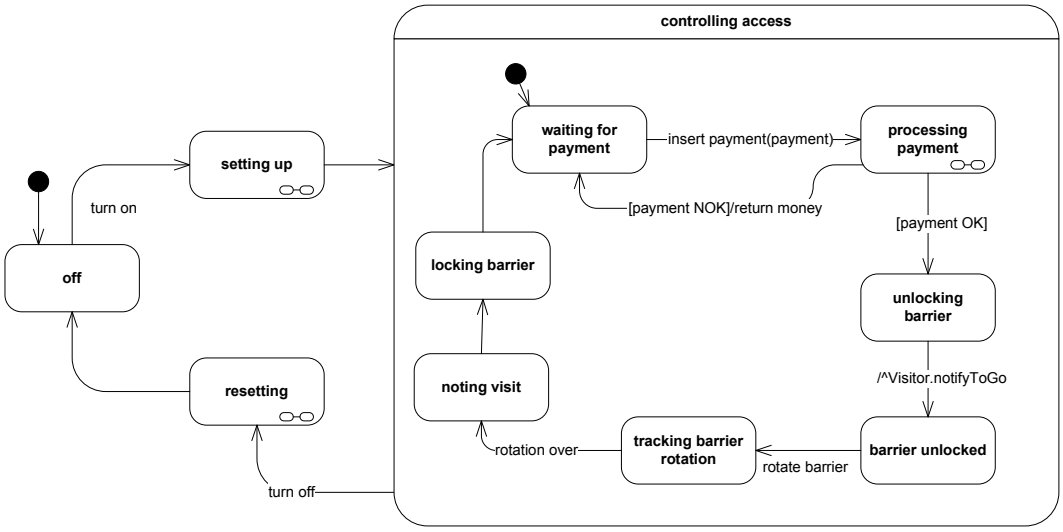


Figure 4.23: Final Unified UC Statechart

4.3.5 Step 5

The fifth step is to **perform conceptual analysis of the unified UC statechart**, i.e., assign activities and data from the unified UC statechart to different concepts.

This step can that can be done in many ways, and the way the students completed their exercise is described in [89]. Figure 4.24 shows the final DM the students produced for the Turnstile CBS within the allocated tutorial time.

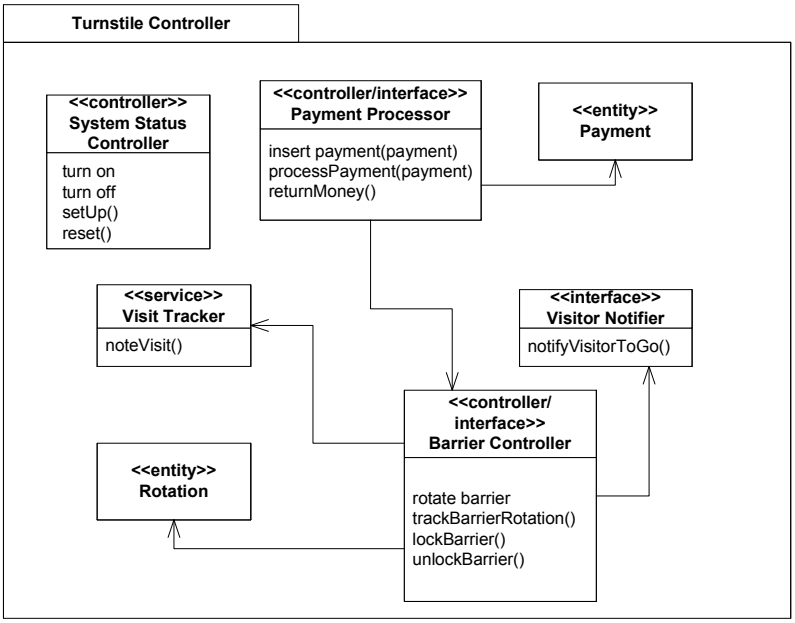


Figure 4.24: Final Turnstile DM

Appendix B shows other diagrams that were specified during the tutorial.

4.4 Comparison of Old and New DMs for the Turnstile CBS

The crucial part of and the novelty in UCUM is the building of a unified UC statechart for a CBS's domain prior to beginning conceptual analysis. The unified

UC statechart is constructed by analyzing and integration into a growing unified UC statechart each of the UCs of the CBS. The DM is then constructed primarily by analyzing the activities and data captured in the unified UC statechart rather than working from UCs. The DM constructed from a unified UC statechart is significantly different from that constructed from UCs.

Figure 4.25 shows the DM called the "original DM" that appears in the example SRS at the course website [4]. This DM is compared to that of Figure 4.24, called the "new DM". The original DM has 5 concepts, while the new DM has 7 concepts. The difference in the number of concepts is probably not significant for this small CBS. In particular, System Status Controller in the new DM is questionable and its removal would leave the new DM with only 6 concepts. However, the qualitative difference between these two DMs is significant.

The main difference between the two DMs is that they have **no single concept in common**! The closest there is to a common concept is that the Turnstile from the original DM corresponds to the Turnstile Controller from the new DM. However, the Turnstile Controller represents the CBS as a whole and is not itself a concept. Why is there such a large difference between these diagrams?

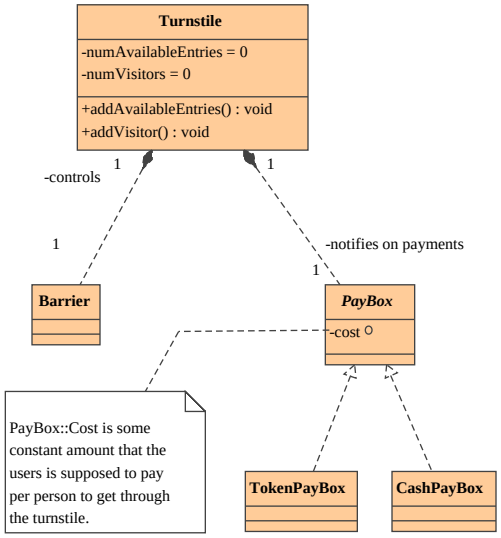


Figure 4.25: Original Turnstile DM [4]

The original DM has three main concepts: Turnstile, Barrier, and PayBox. In fact, the author of the original DM provided an even better diagram, that shown in Figure 4.26, that helps analyze these concepts. The main problem with the concepts in the original DM is that they add nothing to what is shown in even more detail in the physical Turnstile diagram shown in Figure 4.26. In fact, the original DM shows even less detail since the Housing was not captured. Since the original DM was produced through conceptual analysis of UCs, it is not clear if this conceptual analysis of UCs added anything because the original DM shows even less information than what was known about concepts even *before* writing these UCs!

Perhaps, one could blame the author of the original DM for not being able to perform a proper conceptual analysis of UCs. However, then one must blame also

the author of this thesis, numerous teaching assistants, the professors who teach the course, and hundreds of students who took the course who were not able to spot the problem with this DM! In fact, even today, when the author attempts to construct a DM by doing conceptual analysis of the original UCs, the author is not able to come up with a DM different from the original DM.

An understanding of the weaknesses of the original DM led the author to question the very idea of conceptual analysis of UCs and whether it brings any benefit.

Of course, the original DM has two additional concepts, which represent two different kinds of PayBox: TokenPayBox and CashPayBox, but they do not add any functional information to the DM. Also the students were unable to trace to their origins. The students therefore deduced that, probably, the original author added them for some secondary reason, such as to demonstrate the use of inheritance — the original Turnstile SRS was produced for educational purposes.

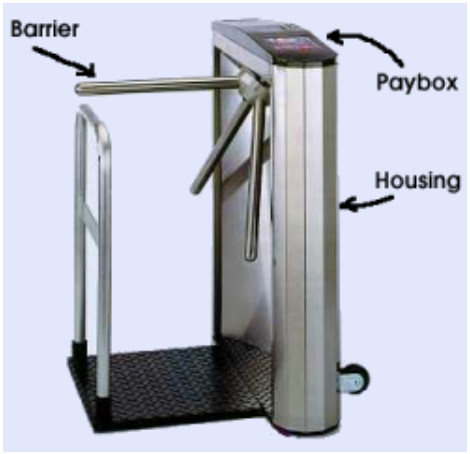


Figure 4.26: Turnstile [4]

On the other hand, following UCUM produces a context diagram, shown in

Figure 4.27, that contains all the concepts shown in the original DM. The students started with the context diagram from the original SRS, but it evolved into a new context diagram that includes the Turnstile's physical components that appeared in the original DM. The new context diagram has captured also the Switch device that was missing in the original DM.

Therefore, this simple case study has shown again the difficulty of trying to identify concepts from UCs, as discussed in Section 2.2. Rather than describing what is *inside of the CBS's domain*, the original DM had the CBS itself represented as a concept inside a DM of the CBS's domain. The reason that the original DM's author was not able to specify what is *inside this CBS* was most likely the basic property that UCs *hide* what is *inside the CBS's domain*, because they show the CBS's domain as a *black box*.

In UCUM, integrating UCs to build a unified UC statechart forces finding and defining the *CBS's boundary*. Finding Turnstile CBS's boundary resulted in Turnstile Controller being clearly identified as the *CBS under consideration* and being included in the *context diagram* rather than in the DM.

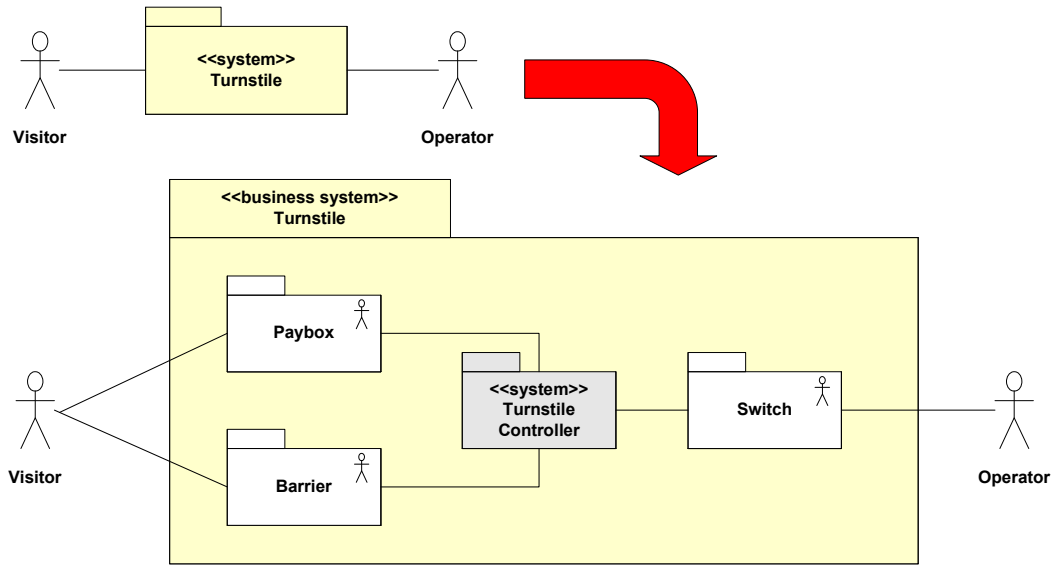


Figure 4.27: System Boundary Redefinition

While the original DM consists of concepts representing *physical devices* and the overall system, the new DM consists of 5 functional concepts that integrate related activities from the unified UC statechart, exhibiting high cohesion and low coupling, and 2 data entities that the CBS has to keep track of. All of these concepts are internal to the Turnstile Controller. Capturing them conforms to the purpose of a DM to show concepts that exist inside the CBS.

4.5 Summary

This chapter describes unified UC statecharts and how they fit within UCUM. It describes the unified UC statechart construction that the students and the author did together as part of tutorials on unified UC statecharts and UCUM in order to

prepare students for their project work. These tutorials gave students insight into modeling and the differences between traditional OOA and UCUM. The analysis of the exercises in CS N3TS showed how UCUM helped capture concepts that are *internal* to the CBS's domain to help construct DMs of higher quality than the DM in the specification document from which the exercise started.

The next chapter evaluates the effects of UCUM and unified UC statecharts on DMs for larger systems, and evaluates their impact on the semantic similarity of the independently specified DMs of the same domain.

Chapter 5

Evaluation

This chapter discusses four case studies. The first two are concerned with DMs produced without the use of UCUM, and the second two are concerned with DMs studies were produced *with* the use of UCUM. The first case study, named "CS O3ES", is of 3 small-sized elevator systems DMs; it is discussed more detail in Appendix A. The second case study, named "CS O31VS", is of 31 term-long group projects in CS445 course [85] that resulted in 31 large-sized specifications of a VoIP system and its IMS. The third case study, named "CS N12ES", is of 12 individual term-long projects in CS846 course [1] that resulted in 12 medium-sized specifications of the controller for a two-elevator system in a low-rise building. The fourth case study, named "CS N46VS", is of 46 term-long group projects in CS445 course [85] that resulted in 46 large-sized specifications of a VoIP system and its IMS. This fourth case study consists of two sets of DMs produced over two terms, 34 DMs produced in one term and 12 DMs produced in

another, later, term. The subcase study of the 34 DMs is named "CS N34VS" and the subcase study of the 12 DMs is named "CS N12VS".

Section 5.1 describes the Elevator System Case Study, CS N12ES. Section 5.2 describes the VoIP Case Studies, CS O31VS and CS N46VS. Section 5.3 generalizes the case studies into a procedure that can be followed by any researcher wishing to replicate the case studies. Section 5.4 describes conclusions and lessons learned from the case studies. Section 5.5 discusses counter indications reported by other experiments. Section 5.6 summarizes the conclusions.

5.1 Elevator System Case Study: CS N12ES

CS N12ES is about 12 DMs for the controller for a two-elevator system in a low-rise building, a medium-sized CBS. Each DM 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 and its behavior, to ensure that all the students had a common starting point. Moreover, much of the ambiguity, which might exist in a fictitious project description, could be resolved by observing the actual elevators' behavior.

Each student handed in two partial SRSs before handing in his final SRS. Each SRS was required to show a set of specific artifacts. The first partial SRS had to show the initial set of UCs for the CBS. Each student was allowed to see all students' sets of UCs before handing in his second partial SRS so that he could have as good a set of UCs as possible before constructing his 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 [1].

Table 5.1 shows a statistical analysis of the numbers of entities found in the context diagrams (CDs) and the DMs. An entity in a CD is called an *actor*, and an entity in a DM is called a *concept*.

A total of 196 original discovered actors appear in the CDs of CS N12ES. After removing syntactic duplicates, 143 unique actors remain. After removing semantic duplicates, i.e., after determining what each actor represents and collapsing semantically similar actors into one, 45 unique actors remain. The maximum, minimum, average, and median numbers of original discovered actors per CD are 25, 10, 16, and 16 respectively.

A total of 213 original discovered concepts appear in the DMs of CS N12ES. After removing syntactic duplicates, 161 unique concepts remain. After removing semantic duplicates, 59 unique concepts remain. The maximum, minimum, average, and median numbers of original discovered concepts per DM are 36, 7, 18, and 18 respectively.

In total, 409 original discovered entities, both actors and concepts, appear in the CDs and DMs of CS N12ES. A total of 285 syntactically unique entities and 104 semantically unique entities appear in CS N12ES. The maximum, minimum, average, and median numbers of original discovered entities per SRS are 48, 20, 34, and 34 respectively.

In the rest of this chapter, "discovered" means "original discovered".

The SRS with the most discovered actors does not have the most discovered

	# of Actors	# of Concepts	Total #
Original, raw, concepts	196	213	409
Syntactically unique concepts	143	161	285
Semantically unique concepts	45	59	104
Maximum	25	36	48
Minimum	10	7	20
Average	16	18	34
Median	16	18	34

Table 5.1: CS N12ES Statistics

	# of Actors	# of Concepts
SRS 1	25	18
SRS 2	16	18
SRS 3	16	12
SRS 4	16	17
SRS 5	19	24
SRS 6	10	10
SRS 7	14	7
SRS 8	23	20
SRS 9	17	22
SRS 10	13	10
SRS 11	15	19
SRS 12	12	36

Table 5.2: CS N12ES Total Number of Actors and Concepts per SRS

concepts and vice versa. The CD with most discovered actors, 25, has 18 discovered concepts; and the DM with most discovered concepts, 36, has 12 discovered actors. Table 5.2 shows the numbers of discovered actors and concepts for each SRS. There is one SRS with an equal number of discovered actors and concepts. There are 5 SRSs whose number of discovered actors is larger than the number of discovered concepts. There are 6 SRSs whose number of discovered concepts is larger than the number of discovered actors.

The overall patterns of the numbers of discovered actors and concepts in the

SRSs of CS O3ES and CS N12ES are similar. The large variation of the numbers of the discovered actors in the CDs of CS N12ES is surprising considering that each student had the opportunity to see and use the same elevators. Of course, the variation might have occurred because not all actors are visible. Another big difference between the SRSs of CS O3ES and CS N12ES is the much larger number of discovered concepts in the DMs than discovered actors in the CDs of CS N12ES compared to the much larger number of discovered actors in the CDs than discovered concepts in the DMs of CS O3ES. These issues are discussed in the next subsection.

5.1.1 Elevator System Actors

In Table 5.3, each actor *A* has a row. The row for *A* is divided into 4 columns. The first column contains *A*'s name. The second column is divided into 12 subcolumns, one for each CD *cd*. At the bottom of the table, the subcolumn for *cd* contains the number of semantically unique actors in *cd*. Row *A*'s entry for the subcolumn for *cd* is black if and only if actor *A* appears in *cd*. The third column contains the number of CDs that have actor *A*. This number should be the number of black subcolumns in the second column of the same row. The fourth column contains the percentage that the number in the third column is of 12.

The subcolumns of the second column are laid out from left to right in the order of decreasing numbers of actors per CD, and the rows are laid out from top to bottom in the order of decreasing numbers of CDs per actor.

Consider the 3 CDs from CS O3ES and the 12 CDs from CS N12ES. These

Actors	CS N12ES Domain Model Actors												# of CDs	% of CDs
	cd1	cd2	cd3	cd4	cd5	cd6	cd7	cd8	cd9	cd10	cd11	cd12		
passenger	■	■	■	■	■	■	■	■	■	■		■	11	92
elevator engine	■	■	■	■	■	■				■	■	■	9	75
floor number display	■	■	■	■	■		■	■	■		■		9	75
mode switch	■	■		■	■	■	■	■	■			■	9	75
operator	■	■	■		■	■	■	■	■			■	9	75
position sensor	■	■	■	■	■	■		■		■	■		9	75
elevator system	■		■	■		■		■	■		■	■	8	67
floor request button	■			■		■	■	■	■	■	■		8	67
alarm button	■	■				■	■	■	■				7	58
button panel (external)	■	■	■			■	■					■	7	58
elevator cab	■	■	■			■		■		■	■		7	58
emergency button	■			■	■		■	■	■			■	7	58
floor button	■	■		■				■	■	■			7	58
load sensor	■	■	■				■	■		■	■		7	58
power switch		■	■		■	■			■		■	■	7	58
door sensor		■	■	■		■					■		5	42
fire system		■	■	■		■			■				5	42
inner door	■		■	■	■						■		5	42
open door button	■				■		■	■		■			5	42
alarm		■	■	■									4	33
button panel (internal)	■		■			■						■	4	33
direction indicator	■			■	■			■					4	33
door			■				■	■		■			4	33
door opening device	■	■				■						■	4	33
door timer				■			■			■			3	25
elevator control room		■	■				■						3	25
outer door			■		■						■		3	25
close door button									■	■			2	17
emergency stop		■				■							2	17
floor		■									■		2	17
floor number display (external)		■	■										2	17
building security monitoring system				■									1	8
door open sensor		■											1	8
elevator shaft	■												1	8
emergency bell	■												1	8
emergency phone						■							1	8
light										■			1	8
load bell	■												1	8
machine room	■												1	8
moving timer										■			1	8
rope	■												1	8
service request interface									■				1	8
sheave	■												1	8
space sensor							■						1	8
stop request indicators									■				1	8
# of Actors	26	31	24	26	18	19	20	26	21	23	24	16		

Table 5.3: CS N12ES Actors

two sets of CDs share 15 semantically unique actors. The CDs of CS N12ES have 30 semantically unique actors that do not appear in the CDs of CS O3ES, and the CDs of CS O3ES have 9 semantically unique actors that do not appear in the CDs of CS N12ES. The 15 common actors are: button panel (internal), close door button, door, door opening device, elevator cab, elevator engine, elevator shaft, elevator system, floor, floor button, floor number display, floor request button, inner door, outer door, and passenger.

These actors are *external* entities discovered through observation and domain knowledge. The sparsity of Table 5.3 and the inconsistency by which actors were captured in the CDs of CS N12ES confirms the results of CS O3ES about the inherent difficulty of discovering actors even when, as in CS N12ES, most of actors were *visible* to the students in an already built and deployed elevator!

An important difference is that each actor in each of the CDs of CS O3ES appears in a CD, while each actor in each of the CDs of CS N12ES appears in a *CD*. The source of this difference is the effect of having built unified UC statecharts first in showing a clear boundary for the elevator CBS. In each DM of CS N12ES, the Elevator Controller System boundary was clearly specified, apparently making it easier for students to clearly distinguish actors from concepts.

In order to help students see the actors that were not visible in the already deployed elevators, all students were directed to the article "How Elevators Work" [2], and were suggested to use it for *background domain knowledge* and for finding the concepts that they could not observe directly. Having the common background reading material and having the ability to see the other students' first partial SRSs

removed the problem of not being able to see the internals of the already deployed elevators. The common reading and the common agreement that the installed elevator is a *roped elevator* seems to have removed much of the ambiguity on what external hardware devices might act as actors to the Elevator Controller System.

5.1.2 Elevator System Concepts

In Table 5.4, each concept c has a row. The row for c is divided into 4 columns. The first column contains c's name. The second column is divided into 12 subcolumns, one for each DM d. At the bottom of the table, the subcolumn for d contains the number of semantically unique concepts in d. Row c's entry for the subcolumn for d is black if and only if concept c appears in d. The third column contains the number of DMs that have concept c. This number should be the number of black subcolumns in the second column of the same row. The fourth column contains the percentage that the number in the third column is of 12.

The subcolumns of the second column are laid out from left to right in the order of decreasing numbers of concepts per DM, and the rows are laid out from top to bottom in the order of decreasing numbers of DMs with the concept.

Consider the DMs from CS O3ES and the DMs from CS N12ES. These two sets of DMs share 8 semantically unique concepts. The DMs of CS N12ES have 51 semantically unique concepts that do not appear in the DMs of CS O3ES, and the DMs of CS O3ES have 12 semantically unique concepts that do not appear in the DMs of CS N12ES. The 8 common actors are: cab controller (appears as elevator controller in each of the DMs of CS O3ES), current floor, designated floor,

Concepts	CS N12ES Domain Model Concepts												# of CDs	% of CDs
	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12		
cab controller	■	■	■	■	■	■	■	■	■	■	■	■	12	100
elevator controller system	■	■	■	■	■	■	■	■	■	■	■	■	12	100
door controller	■	■	■	■		■	■	■	■	■		■	10	83
request processor	■	■	■	■		■	■	■	■	■	■		10	83
floor number display	■	■	■	■	■			■	■	■		■	9	75
system status controller	■		■	■	■	■	■		■	■		■	9	75
current floor	■	■	■	■		■	■		■				7	58
scheduler		■		■	■			■		■	■	■	7	58
alarm controller	■	■	■				■	■				■	6	50
floor sensor	■		■	■	■						■		6	50
mode controller	■	■	■		■			■		■			6	50
request	■	■	■				■		■		■		6	50
direction	■	■		■		■			■				5	42
emergency processor	■	■				■		■					5	42
alarm		■			■		■				■		4	33
door sensor	■		■		■						■		4	33
elevator cab	■	■		■		■							4	33
pending queue	■			■	■								4	33
timer	■	■			■		■						4	33
designated floor	■		■				■						3	25
door opening device	■	■			■								3	25
fire alarm controller	■	■								■			3	25
load sensor	■		■		■								3	25
cab status				■	■								2	17
direction indicator				■					■				2	17
door timer	■					■							2	17
load						■	■						2	17
operation mode	■		■										2	17
recall floor	■			■									2	17
security service		■					■						2	17
service switch handler						■		■					2	17
time period	■								■				2	17
bottom floor	■												1	8
button status	■												1	8
cab speed						■							1	8
cab status manager				■									1	8
car position indicator					■								1	8
car selector		■											1	8
constraint processor								■					1	8
elevator button					■								1	8
elevator input pane processor								■					1	8
elevator recaller			■										1	8
fire alarm	■												1	8
floor button					■								1	8
floor request logger						■							1	8
hallway input panel processor								■					1	8
hardware controller			■										1	8
load alarm							■						1	8
load calculator										■			1	8
motion sensor							■						1	8
pending queue manager									■				1	8
recall controller		■											1	8
request scheduler	■												1	8
ring bell request processor													1	8
sensors monitor			■										1	8
service controller			■										1	8
summon request logger						■							1	8
top floor	■												1	8
verification processor								■					1	8
# of Concepts	31	20	19	17	17	16	16	14	12	10	8	7		

Table 5.4: CS N12ES Concepts

direction, door timer, pending queue, request, and time period.

Each of the concepts cab controller, elevator controller system, door controller, request processor, floor number display, and system status controller, appears in more than 75% of the DMs of CS N12ES. These 6 concepts amount to 10% of the overall number of semantically unique concepts. If we consider only each concept that appears in 2 or more DMs, there are 32 semantically unique concepts. The number of concepts such that each appears in at least half of the DMs is 12, which is 20% of the total number of concepts. These numbers amount to an improvement in completeness of the list of concepts in the DMs in the DMs of CS N12ES over those in the DMs of CS O3ES. However, because of the small number of DMs in CS O3ES, it is hard to generalize this observation. The next section presents a more generalizable quantitative comparison.

On the other hand, the two sets of elevator DMs have provided qualitative insights into the impact of building unified UC statecharts and detailed functional decompositions on the later building of DMs. In addition to the clear definition of a CBS's boundary and the separation of concepts from actors, the author observed following four characteristics of the CS N12ES DMs:

- a large number of *functional* concepts, i.e., *processors*,
- a clear definition of *interface* concepts,
- an insufficient number of *data* concepts, and
- a lack of *inheritance*.

Out of the 32 semantically unique concepts that appear in 2 or more DMs, each of 14 concepts is of *processor* type, i.e., responsible for functional processing and possibly interfacing with external entities. Each of 8 additional concepts is of *interface* type, i.e., responsible for interfacing with external entities. Together, these 14 processor concepts and these 8 interface concepts amount to two thirds of all 32 semantically unique concepts.

Processor concepts thus make a majority of the concepts captured. Considering the extent to which the students performed *functional* analysis and decomposition prior to starting to build DMs from unified UC statecharts, that so many concepts they found are processor concepts is not surprising. Also the medium number of interface concepts students found is not surprising given the impact of building unified UC statecharts had on defining the proper CBS's boundary and in identifying actors.

Ten out of the 32 semantically unique concepts that appear in more than 2 DMs are *data* concepts. Despite that *data* concepts appeared as a large fraction of these 32 semantically unique concepts in all DMs together, each DM has a very low occurrence *data* concepts, as is shown in Table 5.5. This low occurrence of data concepts is rather surprising since the students were instructed to systematically capture *events* and *event parameters* as concepts as part of UCUM. Among the concepts that appear in only 1 DM, 4 are *data* concepts, 8 are *interface* concepts, and 15 are *processor* concepts. In spite of the overall improvement in data concept capture in the DMs of CS N12ES over data capture in the DMs of CS O3ES, the low extent to which data concepts were captured is disappointing.

Processor Concepts	# of DMs	Data Concepts	# of DMs	Interface Concepts	# of DMs
cab controller	12	current floor	7	floor number display	9
elevator controller system	12	request	6	floor sensor	6
door controller	10	direction	5	alarm	4
request processor	10	pending queue	4	door sensor	4
system status controller	9	designated floor	3	elevator cab	4
scheduler	7	cab status	2	door opening device	3
alarm controller	6	load	2	direction indicator	2
mode controller	6	operation mode	2	security service	2
emergency processor	5	recall floor	2	-	-
timer	4	time period	2	-	-
fire alarm controller	3	-	-	-	-
load sensor	3	-	-	-	-
door timer	2	-	-	-	-
service switch handler	2	-	-	-	-

Table 5.5: CS N12ES Concept Types

Finally, none of the students used *inheritance* in their elevator DMs. The author originally thought that this was due to the low number of *data* concepts. In retrospect, the lack of use of inheritance is likely primarily due to the students' and the author's inability to see the relevance from the *analysis* perspective of *inheritance* relationships among concepts. In the DMs of CS O3ES, inheritance was used to capture relationships among *concrete* concepts such as floor button and the *abstract* concepts such as button. The author was not able to see any net benefit to inheritance in the elevator CBS's DMs: The value it adds to a DM is reduced by the amount of *diagram noise* that it adds to the DM. In the DMs of CS N12ES, there was not a single use of inheritance, despite its emphasis in the academic program of which CS846 is a part and the students' backgrounds. Thus, the author observes that the reason for the lack of inheritance was the focus on conceptual analysis of unified UC statechart's; most likely the students were too busy searching for activities and assigning them to *concrete* concepts and had no time for analysis of additional *abstract* concepts and their relationships to concrete

concepts.

5.2 VoIP Case Studies: CS O31VS and CS N46VS

CS O31VS is about 31 DMs for a VoIP system and its IMS produced *without* the use of UCUM, and CS N46VS is about 46 DMs for the same large-sized CBS produced *with* the use of UCUM. The production of the 31 SRSs of CS O31VS was carried out over one term, and was done before the production of the 46 SRSs of CS N46VS even began. Each SRS was produced by a group of 3 or 4 primarily undergraduate CS445 students working together. The production of the 46 SRSs of CS N46VS was carried out over two terms, one producing the 34 SRSs of CS N34VS and the other producing the 12 SRSs of CS N12VS. 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.[1]

In the CS O31VS and CS N34VS terms, 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 CS N12VS term, the groups were not required to hand the first partial SRS, but each group had to do a formal walkthrough to the TA and course staff demonstrating the UCs they had found so far.

In each term, each successive partial and final SRS was required to show a growing set of specific artifacts. Each group worked independently, and no group

[1]The concept names in this case study are disguised because at the time of publishing the thesis, the analyzed project is still being used in the course from which the data come.

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 description of the CBS can be found at the course website [1].

This section compares the DMs of CS O31VS produced in the term not using UCUM, with the DMs of CS N34VS produced in the term using UCUM, in which unified UC statecharts instead of sequence diagrams are used to help derive DMs from UCs. The course organization and project organization were identical in both of these terms except for the introduction in the later term of unified UC statecharts and UCUM in the lectures and among the required artifacts and methods to be used in the project. Also, the students' backgrounds and the ratio of software engineering, computer science, and electrical engineering students were very similar in both terms. The comparison does not include the DMs of CS N12VS because of the changes in the project organization, the SRS structure, teaching support, and differences in the students' backgrounds in the studied term. Nevertheless, the experiences about the effectiveness of UCUM and the usefulness of unified UC statecharts in the CS N12VS term were identical to the experiences in the CS N34VS term, even though the teaching staff was more familiar with UCUM and could avoid the pitfalls encountered during the first offering of the course with UCUM.

Table 5.6 shows the total number of concepts discovered and captured in all DMs of CS O31VS and CS N34VS. In the CS O31VS DMs, there was a total of 527 discovered concepts. After removing syntactic duplicates, there were 259

	CS O31VS	CS N34VS	CS N34VS w/o DM45
Original, raw, concepts	527	622	-
Syntactically unique concepts	259	312	-
Semantically unique concepts	134	140	110
Maximum	31	45	33
Minimum	8	10	10
Average	17	18	17
Median	16	17	17

Table 5.6: CS O31VS–CS N34VS Statistics

unique concepts. After removing semantic duplicates, there were 134 unique concepts. The maximum, minimum, average, and median were 31, 8, 17, and 16.

In the CS N34VS DMs, there was a total of 622 discovered concepts. After removing syntactic duplicates, there were 312 unique concepts. After removing semantic duplicates, there were 140 unique concepts. The maximum, minimum, average, and median were 45, 10, 18, and 17.

Table 5.6 shows also in the fourth column maximum, minimum, average, and median, for all CS N34VS DMs but one with most concepts. This particular DM with 45 concepts contained a large number of concepts that the author was unable to classify and really understand what they represent. This DM is the subject of a later discussion, and the author will ignore it in the discussion until then, unless the author explicitly mention it as DM45.

The numbers from the two case studies, CS O31VS and CS N34VS, are surprisingly similar. The maximum numbers of concepts per DM are 31 and 33 respectively, the average numbers of concepts per DM are 17 and 17 respectively, and the median numbers of concepts per DM are 16 and 17 respectively.

The similarity in the numbers of concepts in the DMs of CS O31VS and CS

N34VS is surprising considering the large difference in the numbers of concepts in the DMs of CS O3ES and CS N12ES. It is hard to determine what this similarity means without examining the actual concepts in the DMs of CS O31VS and CS N34VS.

5.2.1 CS O31VS: VoIP DMs without UCUM

In Table 5.7, each concept c has a row. The row for c is divided into 4 columns. The first column contains c's name. The second column is divided into 31 subcolumns, one for each DM d. At the bottom of the table, the subcolumn for d contains the number of semantically unique concepts in d. Row c's entry for the subcolumn for d is black if and only if concept c appears in d. The third column contains the number of DMs that have concept c. This number should be the number of black subcolumns in the second column of the same row. The fourth column contains the percentage that the number in the third column is of 31.

The subcolumns of the second column are laid out from left to right in the order of decreasing numbers of concepts per DM, and the rows are laid out from top to bottom in the order of decreasing numbers of DMs per concept. A thick horizontal line separates the concepts that occur in more than 1 DM from the concepts that occur in only 1 DM.

There were 134 discovered concepts all together in the DMs of CS O31VS. No single concept was captured in all DMs, although one concept, user account, was captured in all but 1 DM. Fifty out of 134 concepts, or 37%, appeared in more than 1 DM. These 50 concepts are called *major concepts* of d, $mc(d)$. Twenty-one

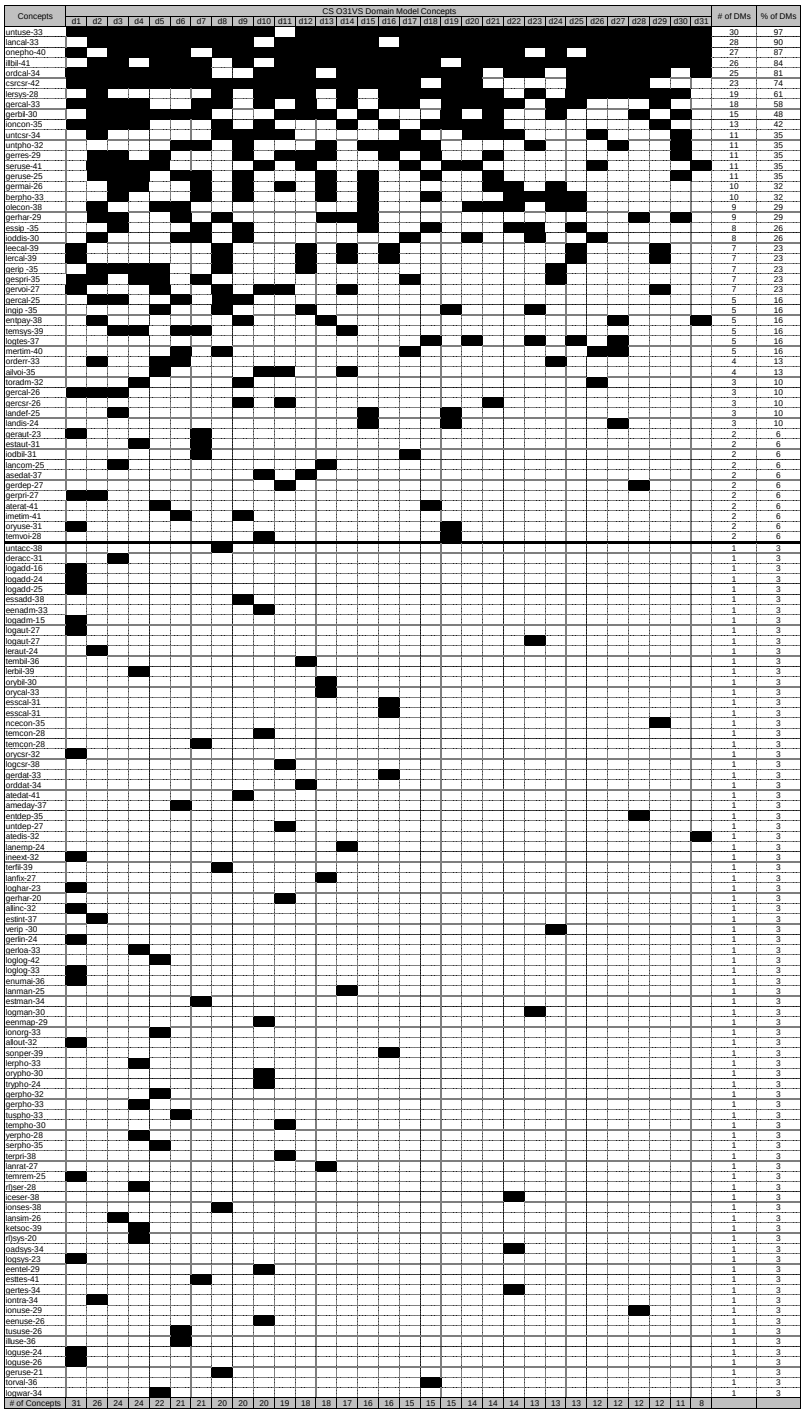


Table 5.7: CS O31VS Concepts

concepts were specified in at least 25% of the DMs. These 21 concepts amount to 16% of the discovered concepts or 42% of *major concepts*. Eight concepts were specified in at least 50% of the DMs. These 8 concepts amount to 6% of the discovered concepts or 16% of the *major concepts*. Five concepts were specified in at least 75% of the DMs. These 5 concepts amount to 4% of the discovered concepts or 10% of the *major concepts*. Eight concepts appeared at least as often as the median number of concepts per DM.

These numbers and just a glance at the Table 5.7 show how little overlap and semantic similarity there is among the DMs of CS O31VS. How does this semantic similarity pattern compare to the semantic similarity pattern among DMs of CS N34VS?

5.2.2 CS N34VS: VoIP DMs with UCUM

In Table 5.8, each concept c has a row. The row for c is divided into 4 columns. The first column contains c's name. The second column is divided into 34 subcolumns, one for each DM d. At the bottom of the table, the subcolumn for d contains the number of semantically unique concepts in d. Row c's entry for the subcolumn for d is black if and only if concept c appears in d. The third column contains the number of DMs that have concept c. This number should be the number of black subcolumns in the second column of the same row. The fourth column contains the percentage that the number in the third column is of 34.

The subcolumns of the second column are laid out from left to right in the order of decreasing numbers of concepts per DM, and the rows are laid out from

top to bottom in the order of decreasing numbers of DMs per concept. A thick horizontal line separates the concepts that occur in more than 1 DM from the concepts that occur in only 1 DM.

There were 140 discovered concepts all together in the DMs of CS N34VS. No single concept was captured in all specifications although one concept, calling plan, appears in all but one DM. Forty seven out of 140 concepts, or 34%, appeared in more than one DM. These 47 concepts are called *major concepts*. Twenty-three concepts were specified in at least 25% of the DMs. These 23 concepts amount to 16% of the discovered concepts or 49% of the *major concepts*. Twelve concepts were specified in at least 50% of the DMs. These 12 concepts amount to 9% of the discovered concepts or 26% of the *major concepts*. Four concepts were specified in at least 75% of the DMs. These 4 concepts amount to 3% of the discovered concepts or 9% of the *major concepts*. Twelve concepts appeared at least as often as the median number of concepts per DM.

In the CS N34VS DMs, 34% of the semantically unique concepts appeared in more than one DM, and in the CS O31VS DMs, 37% of the semantically unique concepts appeared in more than 1 DM, as is shown in Table 5.9, second row, columns 6 and 3, respectively. These data give the impression that the concentration of semantically similar concepts was higher in the CS O31VS DMs than in the CS N34VS DMs. However, considering that the CS N34VS DMs had three more DMs than the CS O31VS DMs and that DM45 contributed a large number of concepts that could not be correlated with those of other DMs, the impression is not clear. Still ignoring DM45, what remains is 45 concepts out of 110, i.e.,

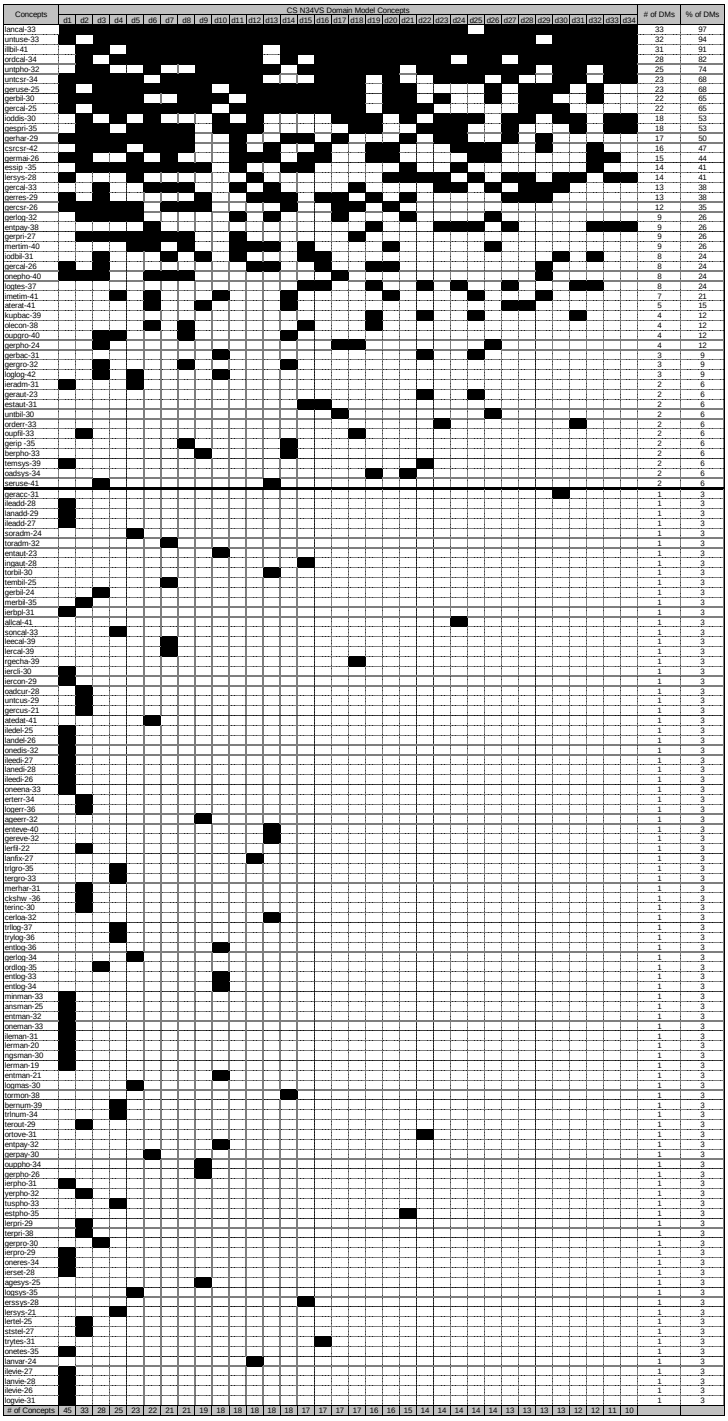


Table 5.8: CS N34VS Concepts

# of DMs	CS O31VS			CS N34VS		
	# of Concepts	% of Concepts	% of Major Concepts	# of Concepts	% of Concepts	% of Major Concepts
>= 2	50	37	100	47	34	100
>= 25%	21	16	42	23	16	49
>= 50%	8	6	16	12	9	26
>= 75%	5	4	10	4	3	9

Table 5.9: CS O31VS–CS N34VS Concept Concentration

41%, appearing in more than 1 DM. The comparison of actual concepts in the next subsection gives a better insight into the difference and similarities between concepts that appear in the DMs of these case studies.

Table 5.9 shows the concentration of semantically similar concepts among the concepts that appear in at least 2, 25%, 50%, and 75% of the DMs in CS O31VS and CS N34VS. The second, third, and fourth column of the table show the number of concepts, that number's percentage out of all semantically unique concepts, and that same number's percentage out of all *major concepts* in the DMs of CS O31VS. The fifth, sixth, and seventh column of the table show the number of concepts, that number's percentage out of all semantically unique concepts, and that same number's percentage out of all *major concepts* in the DMs of CS N34VS.

On the other hand, when the two tables Table 5.7 and Table 5.8 are shown side-by-side as they are in Figure 5.1, then the facts that:

1. there is a bit less white in the mostly black region at the top of Table 5.8 than in the same part of Table 5.7, and

2. there is a bit less black in the mostly white region at the bottom of Table 5.8 than in the same part of Table 5.7,

says that the DMs of the DMs of CS N34VS, described by Table 5.8, are a bit more semantically similar to each other than the DMs of the DMs of CS O31VS, described by Table 5.7.

The third row of Table 5.9 shows that the DMs of CS O31VS had 16% of all semantically unique concepts occurring in at least 25% of the DMs, the DMs of CS N34VS had 16% of all semantically unique concepts occurring in at least 25% of the DMs, the DMs of CS O31VS had 42% of *major concepts* occurring in at least 25% of the DMs, and the DMs of CS N34VS had 49% of *major concepts* occurring in at least 25% of the DMs.

The fourth row of Table 5.9 shows that the DMs of CS O31VS had 6% of all semantically unique concepts occurring in at least 50% of the DMs, the DMs of CS N34VS had 9% of all semantically unique concepts occurring in at least 50% of the DMs, the DMs of CS O31VS had 16% of *major concepts* occurring in at least 50% of the DMs, and the DMs of CS N34VS had 26% of *major concepts* occurring in at least 50% of the DMs.

The fifth row of Table 5.9 shows that the DMs of CS O31VS had 4% of all semantically unique concepts occurring in at least 75% of the DMs, the DMs of CS N34VS had 3% of all semantically unique concepts occurring in at least 75% of the DMs, the DMs of CS O31VS had 10% of *major concepts* occurring in at least 75% of the DMs, and the DMs of CS N34VS had 9% of *major concepts* occurring in at least 75% of the DMs.

Overall, these numbers show that the DMs of CS N34VS had a larger concentration of semantically similar concepts did than the DMs of CS O31VS. More-

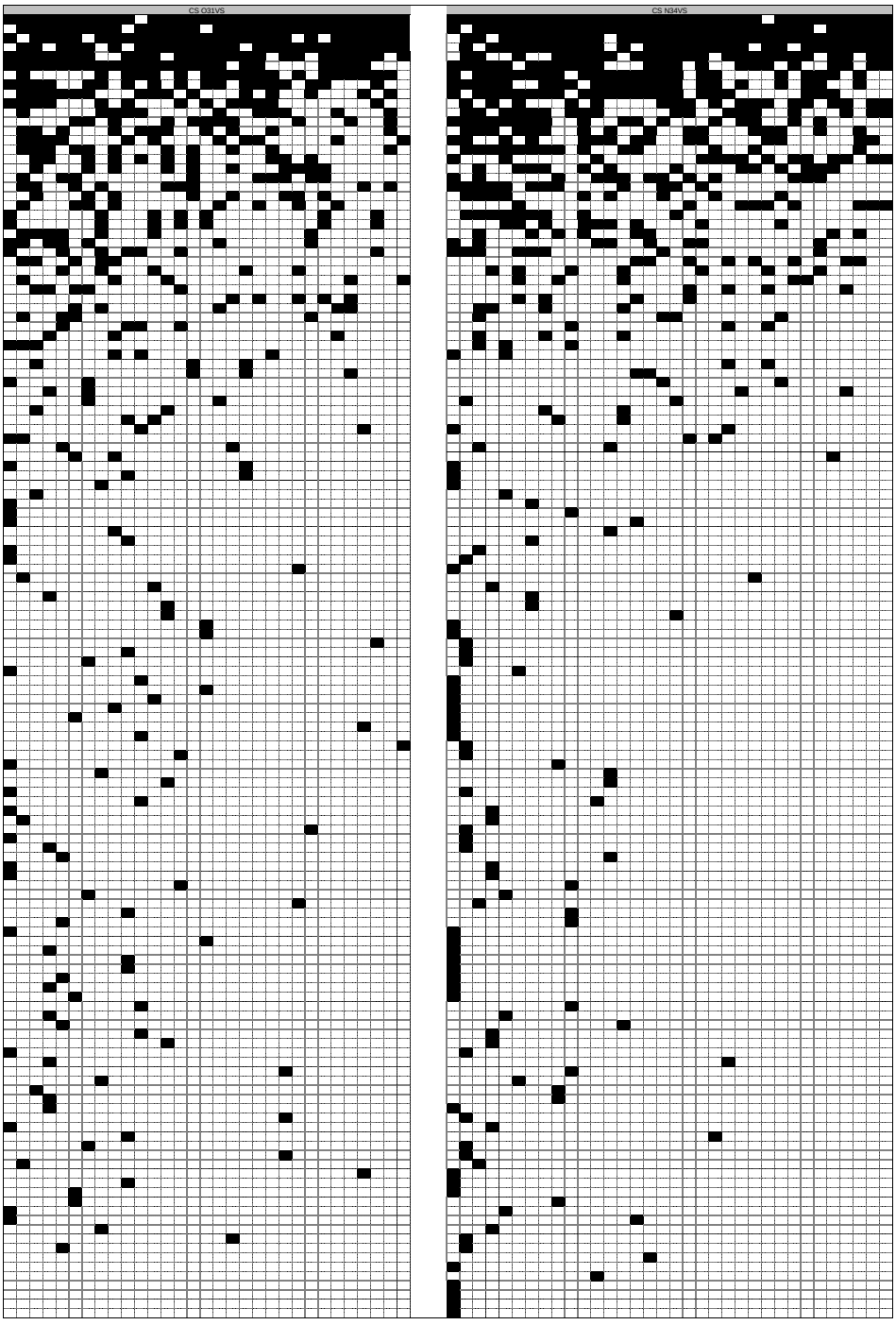


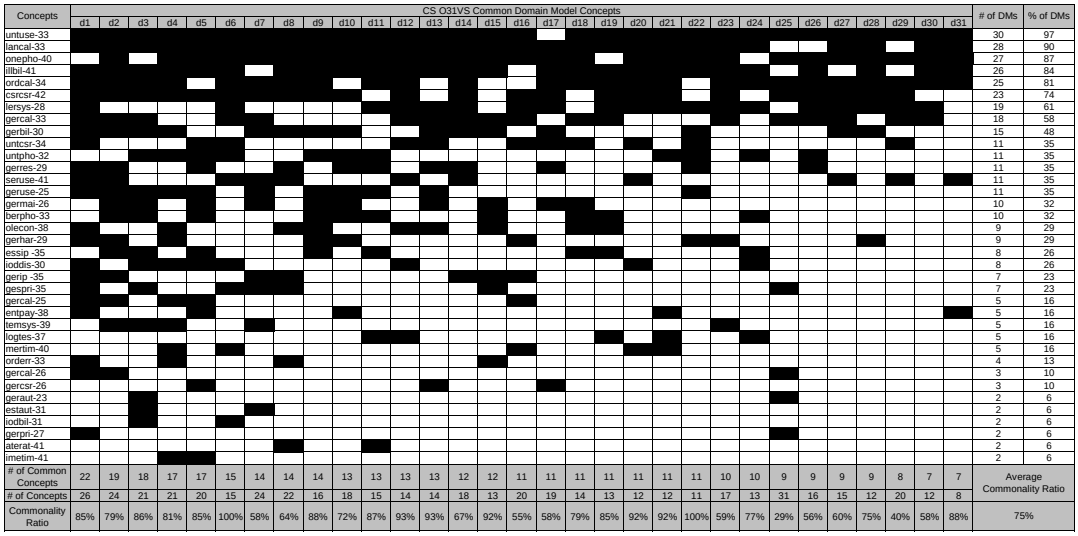
Figure 5.1: CS O31VS–CS N34VS Side-By-Side

over, the DMs of CS N34VS had a larger concentration of semantically similar *major concepts* than did the DMs of CS O31VS. Nevertheless, considering only *major concepts* does not give any indication of which *major concepts* are *common* to the DMs of both case studies. Comparing *common concepts* gives the best measure of which concepts were appearing in the DMs of both case studies, independent of the OOA method used, and how semantically similar their DMs are.

5.2.3 Comparative Evaluation of the DMs in CS O31VS and CS N34VS

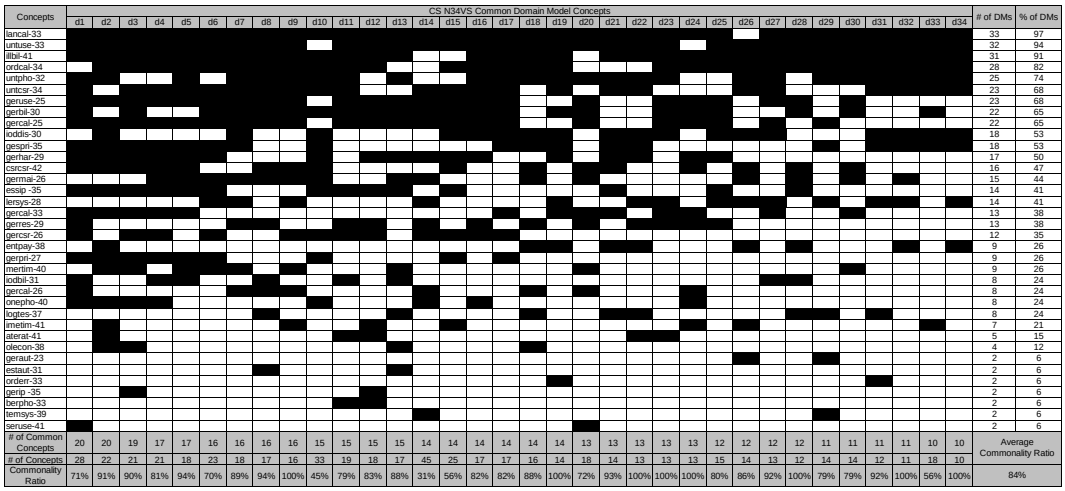
The final comparison of the semantic similarity of the concept sets of the DMs in CS O31VS and CS N34VS considers the 36 concepts that appear in the DMs of *both* case studies, including the DM45 in CS N34VS. Table 5.10 shows the distribution of these 36 *common concepts* in the DMs of CS O31VS, and Table 5.11 shows the distribution of these 36 *common concepts* in CS N34VS. The 36 *common concepts* amount to 72% of the *major concepts* in the DMs in CS O31VS and 77% of the *major concepts* in the DMs in CS N34VS.

The layouts of Tables 5.10 and 5.11 are the same. In each table, each concept c has a row. The row for c is divided into 4 columns. The first column contains c's name. The second column is divided into a number of subcolumns, one for each DM d. Row c's entry for the subcolumn for d is black if and only if concept c appears in d. The third column contains the number of DMs that have concept c.



Concepts	CS O31VS Common Domain Model Concepts																													# of DMs	% of DMs
	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12	d13	d14	d15	d16	d17	d18	d19	d20	d21	d22	d23	d24	d25	d26	d27	d28	d29	d30	d31		
untuse-33																																30	97
lancal-33																																28	90
onepho-40																																27	87
illbil-41																																26	84
ordcal-34																																25	81
csrcsr-42																																23	74
lersys-28																																19	61
gercal-33																																18	58
gerbil-30																																15	48
untcsr-34																																11	35
untpho-32																																11	35
gerres-29																																11	35
seruse-41																																11	35
geruse-25																																10	32
germai-26																																10	32
berpho-33																																9	29
olecon-38																																9	29
gerhar-29																																8	26
essip -35																																8	26
ioddis-30																																7	23
gerip -35																																7	23
gespri-35																																5	16
gercal-25																																5	16
entpay-38																																5	16
temsys-39																																5	16
logtes-37																																5	16
mertim-40																																4	13
orderr-33																																3	10
gercal-26																																3	10
gercsr-26																																2	6
geraut-23																																2	6
estaut-31																																2	6
iodbil-31																																2	6
gerpri-27																																2	6
aterat-41																																2	6
imetim-41																																2	6
# of Common Concepts	22	19	18	17	17	15	14	14	14	13	13	13	13	12	12	11	11	11	11	11	11	11	10	10	9	9	9	9	8	7	7	Average Commonality Ratio	
# of Concepts	26	24	21	21	20	15	24	22	16	18	15	14	14	18	13	20	19	14	13	12	12	11	17	13	31	16	15	12	20	12	8		
Commonality Ratio	85%	79%	86%	81%	85%	100%	58%	64%	88%	72%	87%	93%	93%	67%	92%	55%	58%	79%	85%	92%	92%	100%	59%	77%	29%	56%	60%	75%	40%	58%	88%	75%	

Table 5.10: CS O31VS Common Concepts



Concepts	CS N34VS Common Domain Model Concepts																															# of DMs	% of DMs
	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12	d13	d14	d15	d16	d17	d18	d19	d20	d21	d22	d23	d24	d25	d26	d27	d28	d29	d30	d31	d32	d33	d34		
lancal-33																																			33	97
untuse-33																																			32	94
illbil-41																																			31	91
ordcal-34																																			28	82
untpho-32																																			25	74
untcsr-34																																			23	68
geruse-25																																			23	68
gerbil-30																																			22	65
gercal-25																																			22	65
ioddis-30																																			18	53
gespri-35																																			18	53
gerhar-29																																			17	50
csrcsr-42																																			16	47
germai-26																																			15	44
essip -35																																			14	41
lersys-28																																			14	41
gercal-33																																			13	38
gerres-29																																			13	38
gercsr-26																																			12	35
entpay-38																																			9	26
gerpri-27																																			9	26
mertim-40																																			9	26
iodbil-31																																			8	24
gercal-26																																			8	24
onepho-40																																			8	24
logtes-37																																			8	24
imetim-41																																			7	21
aterat-41																																			5	15
olecon-38																																			4	12
geraut-23																																			2	6
estaut-31																																			2	6
orderr-33																																			2	6
gerip -35																																			2	6
berpho-33																																			2	6
temsys-39																																			2	6
seruse-41																																			2	6
# of Common Concepts	20	20	19	17	17	16	16	16	16	15	15	15	15	14	14	14	14	14	14	13	13	13	13	13	12	12	12	12	11	11	11	11	10	10	Average Commonality Ratio	
# of Concepts	28	22	21	21	18	23	18	17	16	33	19	18	17	45	25	17	17	16	14	18	14	13	13	13	15	14	13	12	14	14	12	11	18	10		
Commonality Ratio	71%	91%	90%	81%	94%	70%	89%	94%	100%	45%	79%	83%	88%	31%	56%	82%	82%	88%	100%	72%	93%	100%	100%	100%	80%	86%	92%	100%	79%	79%	92%	100%	56%	100%	84%	

Table 5.11: CS N34VS Common Concepts

This number should be the number of black subcolumns in the second column of the same row. The fourth column contains the percentage that the number in the third column is of the total number of DMs.

In the third last row of the table, the subcolumn for any d contains the number of *common concepts* in d. In the second last row of the table, the subcolumn for any d contains the number of *all* semantically unique concepts in d. In the last row of the table, the subcolumn for any d contains the *commonality ratio* for d, i.e., the ratio of the number of *common concepts* and the number of *all* semantically unique concepts in d. Finally, the cell at the intersection of the last row and the last column contains the *average commonality ratio* for the DMs of the case study, the average of all the commonality ratios in the last row.

A visual inspection of the data distribution patterns in Tables 5.10 and 5.11 gives an impression similar to the impression that a visual inspection of the data distribution patterns in Tables 5.7 and 5.8 gives. Therefore, additional analysis is needed. Tables 5.10 and 5.11 show that

1. on average, 84% of all concepts in the DMs of CS N34VS and

2. on average, 75% of all concepts in the DMs of CS O31VS

are the concepts *common* to the DMs of both case studies, indicating a higher concentration of *common concepts* in each DM of CS N34VS than in each DM of CS O31VS. Table 5.12 shows that the semantic similarity of *common concepts* in the DMs of CS N34VS is higher than the semantic similarity of *common concepts* in the DMs of CS O31VS.

Metrics	CS O31VS Concepts	CS N34VS Concepts	CS O31VS Common Concepts	CS N34VS Common Concepts
Average	16.97	17.91	12.29	14.06
Standard Deviation	5.07	6.85	3.53	2.6
Standard Error	0.91	1.17	0.63	0.45
Quartile (.75)	20	18.75	14	15.75
Quartile (.25)	13	14	10	12
Interquartile Range	7	4.75	4	3.75

Table 5.12: CS O31VS–CS N34VS Statistics Summary

The average number of the concepts per DM is approximately 5.5% higher for the DMs of CS N34VS than for DMs of CS O31VS, as is shown in the cells in Row 2 and Columns 2 and 3 of Table 5.12. The average number of the *common concepts* per DM is approximately 14.4% higher for the DMs of CS N34VS than for DMs of CS O31VS, as is shown in the cells in Row 2 and Columns 4 and 5 of Table 5.12. The standard error of the average is approximately the same in both sets of data, due to the approximately equal size of the sets of data.

The standard deviation of the average number of the concepts per DM is approximately 35.1% higher for the DMs of CS N34VS than for DMs of CS O31VS, as is shown in the cells in Row 3 and Columns 2 and 3 of Table 5.12. The standard deviation of the average number of the *common concepts* per DM is approximately 35.8% lower for the DMs of CS N34VS than for the DMs of CS O31VS, as is shown in the cells in Row 3 and Columns 4 and 5 of Table 5.12. The author assumed that the standard deviation was higher in the first comparison due to the presence of DM45 concepts. Therefore, the author computed also the interquartile range, which ignores DMs with extremely large and extremely low number of concepts.

The interquartile range of the number of the concepts per DM is approximately

47.4% lower for the DMs of CS N34VS than for DMs of CS O31VS, as is shown in the cells in Row 7 and Columns 2 and 3 of Table 5.12. The interquartile range of the number of *common concepts* per DM is approximately 6.7% lower for the DMs of CS N34VS than for the DMs of CS O31VS, as is shown in the cells in Row 7 and Columns 4 and 5 of Table 5.12. That the interquartile range of the number of *common concepts* per DM is lower in CS N34VS than in CS O31VS for the same set of data leads to the conclusion that the concept concentration was higher in the DMs of CS N34VS than in the DMs of CS O31VS, for both all and the *common concepts*.

In summary, the average number of concepts in the DMs of CS N34VS is 5.5% higher than the average number of concepts in the DMs of CS O31VS. The average number of *common concepts* in the DMs of CS N34VS is 14.4% higher than the average number of *common concepts* in the DMs of CS O31VS. The concept concentration in the DMs of CS N34VS is higher than in the DMs of CS O31VS for all and the *common concepts*. Therefore, the semantic similarity in the captured concepts from one DM to another is higher in the DMs of CS N34VS than in the DMs of CS O31VS. So, the author makes a conservative estimate that the semantic similarity of concepts is **approximately 10% higher** in the DMs of CS N34VS than in the DMs of CS O31VS, based on

- an average increase of 5.5% in the number of concepts per DM of CS N34VS over the number of concepts per DM of CS O31VS
- an average increase of 14.4% in the number of *common concepts* per DM

of CS N34VS over the number of *common concepts* per DM of CS O31VS

- an increase in the capture of *common concepts* per DM from 75% in the DMs of CS O31VS to 84% in the DMs of CS N34VS;
- a narrower data spread for both sets of data; and
- an overall qualitative analysis.

The semantic similarity of the DMs of CS N34VS is 10% higher than the semantic similarity of the DMs in CS O31VS.

It is also important to note that the cost of producing the SRSs in the CS N34VS was approximately 25% higher for the SRSs of CS N34VS than for the SRSs of CS O31VS. That is, teaching students and TAs UCUM required 4 hours that were not originally allocated to the course. Of these 4 hours, 2 were spent teaching specification of unified UC statecharts and 2 were spent teaching how specification of unified UC statecharts fits in the overall OOA process. An additional 1 hour was set aside for a question-and-answer session about the material. The author was responsible for answering students' questions found his workload increased about 30% over that in previous terms, in which UCUM was not used. Also, in each term in CS445, we have each TA report his or her actual workload for the course. 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 UCUM-using term. That is, using any variant of UCUM required about 25% more elicitation effort. Because the course staff had anticipated at the beginning of the UCUM-using term that UCUM might require more work, the course

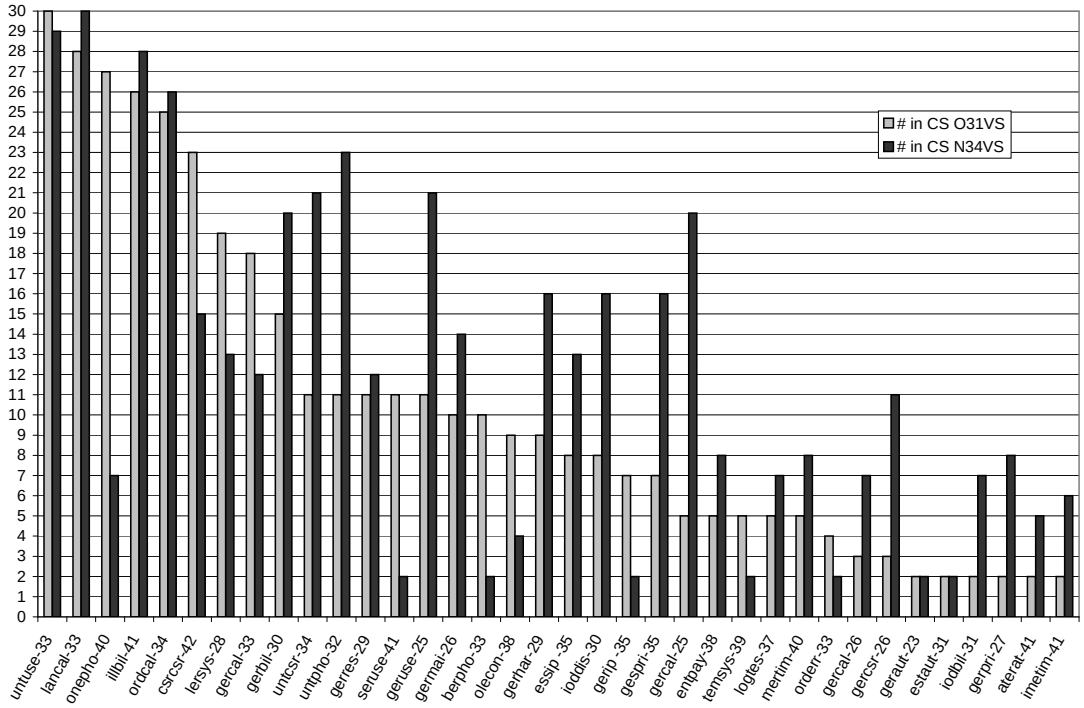


Figure 5.2: CS O31VS–CS N34VS Common Concepts Chart

staff 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 next step is to investigate the qualitative difference in these two case studies. The focus is on the *common* set of concepts that appeared in the DMs of both case studies. The chart in Figure 5.2 is built from the data in Tables 5.10 and 5.11. The number of groups in CS N34VS was scaled down to 31, multiplying by $31/34$, in order to match the number of groups in CS O31VS.

The lower number of a concept c in the DMs of CS N34VS than in the DMs of CS O31VS indicates either that the use of unified UC statecharts and UCUM

prevented *c*'s discovery or that the use of unified UC statecharts and UCUM filtered *c* out of DMs because *c* was outside of the domain's boundary. The same number of *c*s in the DMs of both case studies indicates no change in the effect of the use of unified UC statecharts and UCUM on the discovery of *c*, and a larger number of *c*s in the DMs of CS N34VS than in the DMs of CS O31VS indicates that the use of unified UC statecharts and UCUM helped *c*'s discovery.

In the DMs of CS N34VS there were:

- 11 concepts, or 30.6%, that were captured less often than in the DMs of CS O31VS,
- 23 concepts, or 63.9%, that were captured more often than in the DMs of CS O31VS, and
- 2 concepts, or 5.5%, that were captured as often as in the DMs of CS O31VS.

There are 4 concepts, out of the 11 that appear less often in the DMs of CS O31VS than in the DMs of CS N34VS, that represent external entities, i.e., actors. These concepts are onepho-40, csrcsr-42, seruse-41, and olecon-38. In fact, despite having low number of actors included in DMs, the author was surprised that these concepts have appeared at all in the DMs of CS N34VS as part of the DMs. The existence of some actors in the DMs of the DMs in CS N34VS indicates that building unified UC statecharts facilitates but does not guarantee proper system boundary definition.

The lower number of lersys-28 and gercal-33 concepts in the DMs of CS N34VS than in the DMs of CS O31VS was due to the level of conceptual decomposition and refinement at which DMs in the DMs of CS N34VS were specified. Lersys-28 and gercal-33 are the high-level concepts responsible for the overall control of the IMS and of the call processing part of VoIP system, respectively. Due to the higher degree of decomposition in the DMs of CS N34VS than in the DMs of CS O31VS, these concepts tended to be less explicitly indicated in the DMs of CS N34VS than in the DMs of CS O31VS. Instead, these less explicitly indicated concepts tended to be included indirectly through their components, i.e., the concepts of which they consist.

The temsys-39 concept represents the CBS itself, and in most cases it was captured through the use of package notation to capture all other concepts within the CBS. The author considered capturing of this concept as redundant because a DM, by definition, captures only concepts *within* the CBS's domain, but for some reason, some students felt a need to include this concept explicitly as part of the DM.

For the last three concepts, berpho-33, gerip -35, and orderr-33, the author was not able to explain why they appeared in lower numbers in the DMs of CS N34VS than in the DMs of CS O31VS. One possible explanation is that berpho-33 was a part of untpho-32 which occurred in higher numbers in CS N34VS and thus reduced the need for explicitly capturing berpho-33 as a separate concept. Also, the author was very surprised that the gerip -35 was captured in lower numbers in the DMs of CS N34VS than in the DMs of CS O31VS, in spite of the larger

number of the DMs of CS N34VS than of the DMs of CS O31VS in which essip -35 concept was captured. Finally, orderr-33 was captured in a very low number of DMs in both case studies, so it can be considered as a difficult concept to discover no matter what method is used.

Overall, the chart of Figure 5.2 shows generally larger black bars, indicating a larger number of concepts in the DMs of CS N34VS than in the DMs of CS O31VS, in agreement with the 10% estimate of the increase in the semantic similarity.

There are some similar patterns of qualitative change between the DMs of CS O31VS and the DMs of CS N34VS that are similar to the patterns of qualitative changes between the DMs of CS O3ES and the DMs of CS N12ES. In addition to the improved definition of a domain's boundary and a separation of concepts from actors, it is possible to observe the other four characteristics of UCUM produced DMs already observed and discussed as part of CS N12ES:

- large number of *functional* concepts, i.e., *processors*,
- clear definition of *interface* concepts,
- insufficient number of *data* concepts, and
- lack of *inheritance*.

Finally, DM45 with 30 unclassified concepts, thought the author an important lesson about a very negative impact of the use of unified UC statecharts and UCUM on DMs. This impact is in the specification of any concept that represents only one function of the CBS. Most of the 30 problematic concepts were

of this type. For example, a concept such as add admin profile is nothing but one high-level function of CBS captured as a concept. The author consider this to be incorrect modeling and misuse of DMs and a negative extreme to which detailed behavioral analysis and modeling can lead. Fortunately, only one out of 34 groups went in this direction, so the problem was not widespread, and it can probably be fixed by simply pointing the analysts in the right direction.

5.3 Semantic Similarity Evaluation Steps

This section describes a general procedure for performing a semantic similarity evaluation for an arbitrary pair A and B of sets of DMs, for the benefit of any researcher who wishes to replicate the case studies.

This description is a necessarily incomplete generalization of the procedure followed in the case studies. The procedure followed in the case studies was invented step by step by the author working from the available data with the goal of estimating in which set of DMs were the DMs more semantically similar to each other. Hence the description shows only the steps the author actually took. A researcher who wishes to follow this procedure on another pair of sets of DMs will have to do similar invention if and when he or she encounters a decision in the procedure that goes in a way different from the way in the case case studies.

Suppose that A and B are sets of DMs. Without loss of generality, the purposes of this procedure are

- to decide whether the DMs contained in A are more semantically similar to

each other than the DMs contained in B, and,

- if the models in A are more semantically similar to each other than the models in B, to compute the percentage P by which the models in A are more semantically similar to each other than the models in B.

This procedure has two phases: a data normalization phase, followed by a calculation phase.

There are two tacit assumptions embodied in the procedure.

- All DMs in both sets have been constructed to model the same system. The procedure does not require this assumption, but it makes little sense to compare the semantic similarity of sets of DMs modeling different systems.

- The comparison is between between *two* sets of DMs, because each set of DMs has been created under differing circumstances, such as the use of a different construction method, and the desire is to see if the differing circumstances lead to differing amounts of semantic similarity in the two sets of DMs.

Phase 1: Data Normalization

DMs that are constructed to model the same problem are likely to have significant semantic overlap, since they are modeling the same conceptual space. At the same time, they are also likely to exhibit semantic and syntactic variation from each other, since each is created by a different individual or group. The purpose

of data normalization is to obtain a normalized view of the concepts in the various DMs of one set of DMs, i.e., to ignore trivial differences in the naming of concepts, in order to be able to show how much the DMs in the set differ, in terms of *semantically unique concepts*.

The name of a concept is taken as its value. Within a single DM, any name is unique (e.g., the use of the word "flight" in one correct travel agency DM refers to the same concept throughout that DM). It is also assumed that any name refers to the same concept in different DMs (e.g., "flight" appearing in two different DMs is assumed to refer to the same flight concept). Finally, all differently named concepts from different DMs that capture the same semantic idea are considered to be the same concept (e.g., "flight", "flightInfo", and "flightNumber" might all refer to the same concept).

To construct a single, representative set of semantically unique concepts in a set D of DMs, first construct the set $RawConcepts(D)$ of all concepts in all of the models in D. Because $RawConcepts(D)$ is a set, *syntactically identical concepts*, i.e., those that have the same name, that appear in more than one DM appear only once in $RawConcepts(D)$.

Next, partition $RawConcepts(D)$ into semantic equivalence classes; i.e., within each equivalence class, the elements are considered to represent the same semantic concept even though they have different names (e.g., "flight", flightInfo", flightNumber"). For each equivalence class, one of the members is chosen to be the representative concept name (e.g., "flight"). The determination of which raw concepts belongs to which equivalence class is performed by analyzing each raw con-

cept's name, attributes, methods, and relationships with other concepts.

Next, for each concept c_{raw} in each DM d in D, replace c_{raw} with the representative concept name c_{rep} for the equivalence class of $RawConcepts(D)$ to which c_{raw} belongs.

Finally, some pruning of the DMs is necessary to reduce the importance of concepts that appear in only one or a few of the DMs. A concept is said to be k-significant if it appears in at least k DMs in D. Then remove from all the DMs in D, all of the concepts that are not k-significant. The case studies used 2 as the value of k.

For ease of discussion, assume, from this point on, that a reference to a concept c belonging to a DM d is a representative concept name rather than the raw concept name that might actually appear in the text of d. Moreover, below, $c(d)$ is defined to be the set of representative concepts appearing in d.

This process might sound very complicated, but it is really just a normalization of different names in different DMs in a set of DMs to a single, unambiguous vocabulary consistent across the set, followed by some simple pruning. The manual process of creating the semantic equivalence classes is labor intensive and slow, as well as highly subjective. However, it is also likely to be the more accurate than any automated approach.

Phase 2: Determination of Greater Semantic Similarity

1. Let D be a set of DMs such that $D = A$ or $D = B$.

   In the following, a, b, and d are DMs such that $a \in A$, $b \in B$, and $d \in D$.

2. For any DM $d \in D$, let $c(d)$ be the set of d's representative concepts.

3. For any DM $d \in D$, for some integer $k \geq 1$, let $mc(d) = c(d)$ restricted to the k-significant.

   By extension, for a set of DMs D, define

$$mc(D) = \bigcup_{d \in D} mc(d)$$

4. Let

$$cc(A,B) = mc(A) \cap mc(B)$$

   i.e., the set of concepts that are *common* to A and B.

   Note that $cc(A,B)$ is *not* the same as the intersection of the concepts of all DMs in $A \cup B$; that intersection would be a much smaller set of concepts. Instead, $cc(A,B)$ contains all of the concepts that belong to at least k DMs in A and at least k DMs in B.

5. For $d \in A \cup B$, let

$$cc(d) = c(d) \cap cc(A,B),$$

   the set of *common concepts* relative to A and B that are found in DM d.

6. Define

$$cr(d) = \frac{|cc(d)|}{|c(d)|}$$

   This *commonality ratio* measures the number of concepts common to A and

B found in a DM d and measures it relative to d's size. This number is close to 1 if d is very similar to the set of *common concepts*, and is close to 0 if d is very dissimilar to the set of *common concepts*, i.e., the number is close to 0 if d contains either few *common concepts* or many uncommon concepts.

7. Define
$$avg_{cc}(D) = \frac{\sum_{d \in D} |cc(d)|}{|D|},$$
the average number of concepts common to A and B found in each d in D.

8. Define
$$avg_{cr}(D) = \frac{\sum_{d \in D} (cr(d))}{|D|},$$
the *average commonality ratio* relative to A and B of each d in D.

9. Now, it is possible to say that the DMs of A are more *semantically similar* to each other than are the DMs of B if and only if
$$avg_{cc}(A) > avg_{cc}(B)$$
and
$$avg_{cr}(A) > avg_{cr}(B)$$
i.e., if and only if, among A and B, the average number of *common concepts* found in members of A is higher and the *average commonality ratio* of members of A is higher.

10. Finally, if the DMs of A are more semantically similar to each other than

are the DMs of B define

$$P = avg(avg_{cc}(A)\% - avg_{cc}(B)\%, avg_{cr}(A)\% - avg_{cr}(B)\%),$$

the estimated percentage by which the DMs of A are more semantically similar to each other than are the DMs of B.

Phase 2: Example

Consider two sets of DMs, A and B. Each of A and B happens to have 12 DMs,

$$A = \{a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12\},$$

and

$$B = \{b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b12\}.$$

Each DM in A and in B has its own set of semantically unique concepts.

The first column of each of Tables 5.13 and 5.14 shows the union of all semantically unique concepts in the table's set of DMs, and it happens that

$$c(A) = c(B) = \{c1, c2, c3, c4, c5, c6, c7, c8, c9, c10\}.$$

Each subcolumn of the second column of each of Tables 5.13 and 5.14 shows which semantically unique concepts belong to the DM d named at the head of the subcolumn, e.g., the black cells in the subcolumn for $a3$ of the Table 5.13 indicate

Concepts	A: c(d)											
	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12
c1	■	■	■	■	■	■	■	■	■	■	■	■
c2	■	■	■	■	■	■	■	■	■	■	■	■
c3	■	■	■	■	■	■	■	■	■		■	
c4	■	■	■	■	■	■	■	■			■	■
c5	■	■	■	■		■	■	■	■	■		
c6	■	■	■		■		■	■	■	■	■	
c7	■	■	■	■	■		■				■	
c8			■	■		■		■	■	■		■
c9	■	■		■	■	■			■			
c10	■	■	■		■					■		■
|c(d)|	9	9	9	8	8	7	7	7	7	6	6	5

Table 5.13: DM Set A Concepts

Concepts	B: c(d)											
	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12
c1	■	■	■	■	■	■	■	■	■	■	■	■
c2	■	■	■	■	■	■	■	■	■	■	■	■
c3	■	■	■			■	■	■	■	■	■	■
c4	■	■	■	■	■		■	■	■	■	■	■
c5	■	■	■	■	■	■		■			■	■
c6	■			■	■	■		■	■	■		
c7	■	■	■	■	■	■	■			■	■	■
c8					■							
c9		■										
c10				■								
|c(d)|	7	7	7	7	7	6	6	6	6	6	6	6

Table 5.14: DM Set B Concepts

that

$$c(a3) = \{c1, c2, c3, c4, c5, c6, c7, c8, c10\}.$$

The bottom row of each of Tables 5.13 and 5.14 shows the total number of semantically unique concepts in each DM in the table's set of DMs. Letting $k = 2$,

$$mc(A) = mc(B) = \{c1, c2, c3, c4, c5, c6, c7\}.$$

With $k = 2$, the set of common concepts relative to A and B is

$$cc(A,B) = mc(A) \cap mc(B) = \{c1, c2, c3, c4, c5, c6, c7\}.$$

The common concepts for A and B are shown in Tables 5.15 and 5.16, respectively. The first column of each of Tables 5.15 and 5.16 shows the $cc(A,B)$, i.e., the union of all common concepts in A and B.

Each subcolumn of the second column of each of Tables 5.15 and 5.16 shows which common concepts belong to the DM d named at the head of the subcolumn, e.g., the black cells in the subcolumn $a3$ of the Table 5.15 indicates that

$$cc(a3) = \{c1, c2, c3, c4, c5, c6, c7\}.$$

The third last row of each of Tables 5.15 and 5.16 shows the total number of common concepts, $|cc(d)|$, in the DM d named at the head of the subcolumn.

The second last row of each of Tables 5.15 and 5.16 shows the total number of all semantically unique concepts, $|c(d)|$, in the DM d named at the head of the subcolumn.

The last row of each of Tables 5.15 and 5.16 shows the commonality ratio, $cr(d)$, of the DM d named at the head of the subcolumn,

$$cr(d) = \frac{|cc(d)|}{|c(d)|}.$$

Finally, in each of Tables 5.15 and 5.16, the cell at the intersection of the last column and

Concepts	A: cc(d)												
	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12	
c1													
c2													
c3													
c4													
c5													
c6													
c7													Average
|cc(d)|	7	7	7	6	6	5	7	6	5	4	6	3	5.75
|c(d)|	9	9	9	8	8	7	7	7	7	6	6	5	7.33
cr(d)	78%	78%	78%	75%	75%	71%	100%	86%	71%	67%	100%	60%	78%

Table 5.15: DM Set A Common Concepts

- the third last row shows the average of common concepts

$$avg_{cc}(D) = \frac{\sum_{d \in D} |cc(d)|}{|D|},$$

- the last row shows the average commonality ratio

$$avg_{cr}(D) = \frac{\sum_{d \in D} (cr(d))}{|D|}.$$

.

Therefore, the data say that the DMs of the set B are more semantically similar to each other, and the percentage by which the DMs of the set B are more semantically similar to each other is approximately 12%.

Additional Optional Phase

The above procedure was used in the case studies to determine for a given pair of sets of DMs, in which set are the DMs more semantically similar to each other. The case studies did some additional calculations that served to confirm the deter-

Concepts	B: cc(d)												
	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12	
c1													
c2													
c3													
c4													
c5													
c6													
c7													Average
|cc(d)|	7	6	7	6	6	6	6	6	6	6	6	6	6.17
|c(d)|	7	7	7	7	7	6	6	6	6	6	6	6	6.42
cr(d)	100%	86%	100%	86%	86%	100%	100%	100%	100%	100%	100%	100%	96%

Table 5.16: DM Set B Common Concepts

minations.

- Let $sdev_{con}(D)$ denote the standard deviation in the number of concepts over a set of DMs.
- Let $sdev_{cc}(D)$ denote the standard deviation in the number of *common concepts* relative to A and B over a set of DMs.
- Let $iq_{con}(D)$ denote the interquartile range of the number of concepts in a set of DMs.
- Let $iq_{cc}(D)$ denote the interquartile range of the number of *common concepts* relative to A and B over a set of DMs.

Then there is additional confirmation that the DMs of A are more semantically similar to each other than are the DMs of B if any of the following is true:

$$sdev_{con}(A) < sdev_{con}(B) \quad iq_{con}(A) < iq_{con}(B)$$

$$sdev_{cc}(A) < sdev_{cc}(B) \qquad iq_{cc}(A) < iq_{cc}(B)$$

A more refined estimate of P can be obtained by using any of the following as an additional component of the average defining P:

$$sdev_{con}(B) - sdev_{con}(A) \quad iq_{con}(B) - iq_{con}(A)$$
$$sdev_{cc}(B) - sdev_{cc}(A) \quad iq_{cc}(B) - iq_{cc}(A)$$

5.4 Overall Evaluation and Main Lessons

The thesis hypothesis was "The SDP can be reduced by performing a detailed behavioral analysis before conceptual analysis." The analysis shows that:

> **SDP can be reduced by about 10% by performing a detailed behavioral analysis before conceptual analysis.**

based on

- an average increase of 5.5% in the number of concepts per DM of CS N34VS over the number of concepts per DM of CS O31VS
- an average increase of 14.4% in the number of *common concepts* per DM of CS N34VS over the number of *common concepts* per DM of CS O31VS
- an increase in the capture of *common concepts* per DM from 75% in the DMs of CS O31VS to 84% in the DMs of CS N34VS;
- a narrower data spread for both sets of data; and
- an overall qualitative analysis.

The semantic similarity of the DMs of CS N34VS was higher than the semantic similarity of the DMs in CS O31VS. However, the cost of producing the SRSs

in the CS N34VS was measurably higher than the cost of producing the SRSs in CS O31VS. The cost of producing an SRS was approximately 25% higher for the SRSs of CS N34VS than for the SRSs of CS O31VS.

The two numbers, 10% and 25%, are not comparable and should not be used to estimate increased semantic similarity and UCUM's impact on the overall development process and the CBS being built without taking additional model and process measurements into account. There is no way to estimate the benefits caused by the 10% improvement in the semantic similarity to the downstream development speed, reliability, robustness, and other qualities of the CBS being developed. Also, there is no way to estimate benefits UCUM brings in downstream development in terms of improved reliability, robustness and other qualities for the specified CBS.

Overall, these case studies exposed the following impact of the use of UCUM and extensive functional and behavioral modeling before proceeding with conceptual analysis:

1. better functional analysis and discovery of more requirements,
2. an increase in the semantic similarity patterns,
3. several qualitative changes:
    - larger number of *functional* concepts, i.e., *processors*,
    - clearer definition of *interface* concepts, and
    - lack of *inheritance*, and

4. still insufficient number of *data* concepts.

The relationship between the first two points is of particular importance. Consider detailed functional and behavioral modeling that was done through the use of the unified UC statecharts. This modeling is very similar, and to some extent nothing but, traditional structured analysis [e.g., 27, 100, 99, 98]. So, rather than contradicting each other, combining structured analysis and OOA helped improve the semantic similarity and the quality of DMs in a way summarized in [81]:

> A weakness of OO is that OO methods only build functional models within the objects. There is no place in the methodology to build a complete functional model. While this is not a problem for some applications (e.g., building a software toolset), for large systems, it can lead to missed requirements. Use cases address this problem, but since all use cases cannot be developed, it is still possible to miss requirements until late in the development cycle.

In addition, it is probably this integration of the detailed functional and behavioral analysis with OOA that has resulted in the deeper, more detailed, and "closer to design and architecture" DMs, as observed by independent reviewers [e.g., 69] and some graduate students familiar with other OOA methods who did DMs of CS N12ES [1]. The same reviewers observed that the method itself is more systematic than many scenario-based OOA methods. The author shares this point of view; the DMs produced using UCUM were felt to provide a more solid base for transitioning to design phases than those produced by the OOA methods that had

been used previously.

A large number of and well decomposed *functional* concepts, i.e., *processors*, a clear definition of *interface* concepts, and a lack of *inheritance*, are what the author considers to be the positive effects of doing a detailed behavioral analysis first on the DMs. In the author's opinion, such DMs allow easier transition to OOD activities and models.

Both traditional OOA and UCUM seem unable to expose the *data* concepts in a domain. It seems that neither behavioral analysis nor conceptual analysis are adequate substitutes for the traditional *data* analysis. *Data* analysis should be a necessary component of any analysis method and, in the author's opinion, of a higher priority than conceptual analysis.

In summary, in the author's opinion, conceptual analysis should not be the main analysis activity. It should be only secondary to functional, behavioral, and data analysis. Note, the author considers OOD completely distinct from OOA and self-sufficient, and none of these conclusions apply to OOD.

5.5 Counter Indications

The measured effectiveness of UCUM as an approach in which an analyst is doing process and functional analysis *before* doing conceptual analysis contradicts, at least superficially, the conclusions of a case study by Kabeli and Shoval [53]. They did case studies that show that doing data modeling before doing functional analysis leads to better OO models than doing functional analysis before doing

data modeling. It is not surprising that early case studies produce results that appear to contradict each other. First of all, neither set of case studies is conclusive, and the two sets of case studies addressed different issues, i.e., neither traditional OOA nor UCUM perform explicit *data* analysis, while the subjects of the Kabeli and Shoval study did explicit *data* analysis. Second, there may be yet other parameters, entirely overlooked in each case study, that consistently account for the contradictory conclusions. Only additional, independent, experimentation in the future can resolve this issue.

5.6 Summary

UCUM is different from traditional OOA, and some may not even consider it OOA. Nevertheless, UCUM does suggest what a full solution to the SDP might be — postponing conceptual analysis part of OOA as much as possible. In any case, the main lesson learned from the case studies is that OOA in its current state cannot be used reliably as the primary driver for production of the predictable and complete specifications. No technique that does not deal with the inherent CID of OOA can be expected to solve the SDP. Only a significant paradigm shift can result in the desired improvement. On the other hand, OOA is very useful as an *auxiliary* technique to help discover and refine functional requirements and will most likely be a necessary component of any future method that leads to predictable and complete DMs.

Chapter 6

Background and Related Work

In the early days of the software engineering, most of the methodology research and technology transfer efforts focused on improving programming, as programming was perceived to be the most difficult task in the development of a CBS. The domains supported by early CBSs tended to be relatively small, well understood, and stable. As programming methods and technology have matured and stabilized, the focus shifted to the automation of larger, less understood, and more volatile domains. Today, a typical domain is often so large, so complex, and changes so frequently that it is very difficult, if not impossible, to understand it completely. The inability to understand a domain and to precisely capture all of its goals and requirements risks the creation of a CBS that does not satisfy all the business needs. This non-satisfaction of needs leads, in turn, to dissatisfied customers and users, frequent changes, and other maintenance difficulties.

The size, complexity, and instability of modern domains has engendered a

need for techniques for effectively capturing and understanding business needs and requirements. Often, the effort required to understand a domain exceeds the effort required to build the supporting CBS. This situation is described often in the literature as the *what vs. how problem* [e.g., 51]. Learning *what* to build in a CBS is often more difficult than learning *how* to build the CBS, i.e., correctly understanding a domain is more difficult than building a CBS to support it [e.g., 52].

To help people understand domains, researchers have devised several methods, which can be divided into two main groups. The first group consists of the methods inspired by different programming paradigms. The second group consists of methods that have roots in traditional business analysis.

Programming paradigms have influenced all development stages, even the early ones, such as requirements analysis and design, through the transfer and adoption of different underlying ideas and techniques. For example, structured and OOP resulted in structured [27, 100, 99] and OO [62, 20, 12] analysis and design. This tradition continues with the emergence of newer paradigms such as aspect orientation [80, 56] and agent orientation [75, 57, 74, 35, 16].

Other researchers have applied traditional business analysis techniques during software RE. Goal-driven [24, 72, 94, 10] requirements engineering has proved to be promising for dealing with domain-level requirements for large CBSs. Goal-driven RE has focused on ensuring that software actually fulfills business needs and requirements.

Software requirements analysis techniques that originated from programming

paradigms do not generally conflict with those that originated from business analysis; rather, they complement each other. The main difference between the techniques becomes evident in the requirement abstraction levels for which they are best suited. For example, goal-driven RE techniques are considered well suited to capture domain-level requirements [63], while OO RE techniques are considered well suited to capture product-level requirements [63].

Each of these techniques has its strengths and weaknesses. However, the main difficulties arise during the integration of the domain and product requirements, for example, in moving from domain-level requirements, such as goals, UCs, and features, to OOA artifacts, such as objects, relationships, object features, and attributes. The need to integrate all these artifacts leads to the problem that is the subject of this thesis. It is difficult but essential to establish meaningful and unambiguous relationships among these artifacts.

6.1 Requirement Abstraction Levels

RE is the essential activity in assuring that one builds CBSs that will satisfy stakeholders' goals. As the need for a systematic method to requirements elicitation and specification first became obvious for very large and complex CBSs, most RE research focused on discovery and development of requirements techniques and artifacts that are tailored to support the development of these very large CBSs in the environments with relatively large amounts of resources. Developers of a small CBS, on the other hand, traditionally used an *ad hoc* method to RE due to

the CBS's small size and the developers' unsystematic approach to development. The importance of systematic RE increases dramatically, even for small CBSs, with the introduction of the product-line approaches, customizable software, *etc*.

The other important impact on RE techniques is due to the nature of the development of large CBSs. Traditionally, the typical CBS is developed in-house, where developers work with relatively stable domains, are responsible for the development of the CBS from scratch, and have relatively stable production teams and a large amount of resources. This way of development has led to the dominance of the requirements specification that focuses mainly on product-level requirements such as features [63] and subsequently on low-level requirements and design.

These product-level and low-level requirements are very well studied and widely applied in industrial settings, but the main difficulty is in ensuring that they fulfill the essential business goals. Product-level requirements form a set of features that, combined, are used to achieve the organization's business goals. The success of the overall goal depends on every single one of the features and on the particularities of their interactions. The problem of achieving goals is exacerbated as a result of their frequent changes, which cause a chain reaction of changes in product-level and low-level requirements.

Lauesen has observed that product-level and low-level requirements management is straightforward and changes to them are relatively easy to deal with in practice [63]. Developers can usually sense when these requirements are not correct and do not fit with each other. This ability usually does not work at the higher

levels of abstractions, and it is the responsibility of the business analyst to deal with these higher level requirements. In an occasional case, it is not even possible to estimate how changes in the product-level requirements effect overall goals until the changes are implemented [63].

Requirements are specified either directly or indirectly for many different purposes and as part of many different engineering activities. One classification that is sufficient for the purposes of this work is [63]:

1. *Business-level requirements specification* — Business-level requirements are most often specified indirectly as part of business reengineering activities [42, 25, 92, 84]. The most common concepts that appear at this level are business goals, processes, resources, and rules. For example, for an elevator CBS, a business-level requirement is: "The elevator shall transport passengers and goods from the any floor to any other floor."

2. *Domain-level requirements specification* — Domain-level requirements, as mentioned previously, are most often indirectly specified in the traditional requirements specifications [24]. Newer, more systematic versions of domain-level RE have received a lot of attention recently [15, 18, 73]. Most explicit domain-level requirements are captured and specified for domains, which are becoming increasingly complex and difficult to adequately support by CBSs [19, 37, 36, 71]. The most common concepts that appear at this specification level are user goals, user tasks, domain input, and domain output. A more recent trend is the incorporation of agent-based analysis as part

of domain modeling [75, 57, 74, 35]. For an elevator CBS, an example domain-level requirement is: "The elevator shall be accessible from each floor."

3. *Product-level requirements specification* — Product-level requirement specifications are the most common type of requirement specifications. There is an extensive body of knowledge about them, and most previous research focused on improving the different techniques used to elicit, specify, and validate this type of requirement. The common artifacts and concepts that occur as parts of product-level specifications are features, UCs, functional lists, data input, data output, *etc*. For an elevator CBS, an example product-level requirement is: "The elevator shall accept elevator calls only while stationary."

4. *Design-level requirements specification* — Design-level requirements specification are the requirements that directly constrain the design of a CBS. Much effort has been invested into its standardization through the Unified Modeling Language (UML) [62, 33]. UML artifacts and underlying techniques present the most common types of concepts and techniques used to capture requirements at this level. This level acts as a transition phase between product-level specification and code-level requirement specifications. For an elevator CBS, an example design-level requirement is: "A queue data structure shall be used to store the data for elevator calls"

5. *Code-level requirements specification* — Code-level requirements are usu-

ally specified as part of the programming activity and describe details of low-level algorithm and data structures. This type of specification is that with which most programmers are familiar, as it is inseparable from coding. Code-level requirements focus mainly on implementation-related issues and constraints and are probably the best understood form of requirements specification. For an elevator CBS, an example code-level requirement is: "Due to the timing constraints, function calls to retrieve elevator call data shall be implemented in the C programming language rather than in the Python programming language."

This thesis work spans both *domain-level requirements specification* and *product-level requirements specification*. OOA models are primarily concerned with capturing domain-level concepts and their interactions. OOA models then indirectly drive the discovery and specification of the product-level requirements. OOA models have a dominant role at these two specification levels, and then they are replaced by OOD models as a way to capture design-level requirements and design decisions.

6.2 Domain Models

The main purpose of a DM is to capture the concepts that exist within a CBS's domain. The domain can be seen as consisting of:

- abstract business concepts, and

- computer concepts, including hardware and software.

6.2.1 Business Concepts

A CBS is a part of a larger business system, and serves as a resource to accomplish business's goals. To build a useful DM, we need to study different concepts of the domain. There are four main sets of concepts that describe a domain:

1. *Business Resources* — All concepts, both physical and abstract, that exist inside the business's environment are business resources. They include people, information, different CBSs, and business supplies and products. They participate in the business's processes. A subset of these resources is a source of modeling concepts for the CBS being built. The value of tracking and preserving knowledge about these concepts is that these concepts are used to perform analysis of the CBS's architecture, to track changes to the domain and the CBS from the beginning, and to evaluate how well the CBS reflects current business needs. For an elevator CBS, an example business resource is the *cable* used to pull up the elevator cab.

2. *Business Goals* — The purpose of performing a business activity is to achieve a business goal. A goal can be decomposed into subgoals, and eventually, we reach goals that have to be satisfied directly by the CBS. The study of business goals allows evaluation of how well the CBS satisfies them and how to improve the CBS. An important part of goal-based engineering is the analysis of the changes to the business goals so the CBS will be able to

continue to support future goals and the removal of obsolete goals. For an elevator CBS, an example business goal is *to move passengers and goods from any floor to any other floor*.

3. *Business Processes* — A CBS under development may participate within several business processes in order to help achieve several business goals. UCs describe subprocesses of larger business processes that are automated by the CBS. It is important to understand a business process as it relates its UCs, which in turn relate requirements that the CBS has to satisfy. For an elevator CBS, an example business process is *a passenger's riding of an elevator cab*.

4. *Business Rules* — Business rules are a major source of constraints on CBS. Many a constraint directly influences the CBS's architecture. Therefore, it is important to understand these constraints and keep track of them, for example, to be able to remove architectural limitations imposed by constraints that do not hold any more. For an elevator CBS, an example business rule is *the elevator will not change its direction until it services all previously received calls that lie in the current traveling direction*.

6.2.2 CBS Concepts

The main aspects of a CBS that should be modeled are:

- *System*,

- *subsystems*,
- *modules*,
- *connectors*,
- *processes*,
- *logical processes*, and
- *hardware devices*.

The *System*[1] concept defines the outermost boundary of the CBS under consideration. The *System* serves as a container for all other concepts, and defines the CBS as a resource in the business system.

A *subsystem* is a part of a *System* or a *subsystem*, being an abstraction of actual physical modules, connectors, and processes. It serves as a container and a building block.

A *module* is a basic architectural building block. For example, in the logical view, it represents a concept that occur in domain, and in the implementation view, it represents a code unit. *Modules* are abstractions of basic building blocks of the domain, depending on the development technology used.

A *connector* is an abstraction of a communication mechanism or a channel that exists in a CBS. Its size and complexity vary from a simple procedure call to a connection on the Internet.

[1]Note that this "system" is spelled out with initial uppercase letter to distinguish it from the generic "system" used elsewhere.

A *process* is the execution of software. *Processes* are basic building blocks of a run-time architectural view.

A *logical process* is a white-box UC. The difference between a *logical process* and a regular, black-box UC is that a *logical process* includes activities that are performed within the System rather than just user-visible activities.

A *hardware device* is a concept that occurs in the run-time architectural view, and it represents a physical device that is a part of the CBS.

6.3 OOAD Method

This section describes the parts of a typical OOAD method. Full details about OOAD methods can be found in many OOAD books, [e.g., 62, 41, 28].

6.3.1 Requirements Modeling

Input: Business tasks, UCs, and other business engineering artifacts.

Goal: Break down business-level artifacts in order to capture and define the scope and responsibilities of a CBS that meets the requirements embodied in the input.

Activities: The main requirements modeling activities are:

- Identify main *business goals*, *processes*, *resources*, and *CBS features*.

- Write UC descriptions with a clear identification of the *actors* and the *data* exchanged between the CBS and the environment, and the *context* expressed through pre-conditions, post-conditions and invariants.
- Draw a UC diagram with all the UCs in order to depict the relationships among UCs.

6.3.2 OOA

Input: All Requirements Modeling artifacts.

Goal: Decompose requirements artifacts and build a domain model.

Activities: The main OOA activities are:

- Relate UCs to each other with respect to their main concerns. This produces the first level of the domain's decomposition into related functionality domains, that is, the domain subsystems. Show the decomposition of the UC diagrams using the UML package notation. Repeat this step for any identified domain subsystem.
- For each domain subsystem, from its task and UC descriptions, extract its classes, attributes, and the relationships among them. Show the decomposition using UML-like class notation in what is known as a DM. This step is present in most OOA methods. The author's opinion is that the UC conceptual analysis is not an effective way for developing the DM. Analysts should have good prior domain knowledge,

which should be the main source for domain concepts. Of course, in the absence of knowledge of the domain, even UC conceptual analysis represents a good starting point.

- Extract common concepts from different domain subsystems and allocate them to common domain subsystems.
- Emphasize relationships among data concepts in the DM. Data concepts represent external data with a high probability of having to be used and preserved within the CBS. These data concepts and their relationships constitute the traditional relational part of the DM.
- With UCs, develop the domain-level interaction diagram. The main goal of this step is to define the domain's external interface.
- With UCs, DMs, and the domain-level interaction diagram, for each domain subsystem, develop the domain subsystem interaction diagrams. The main goal of this step is to define the domain subsystems' interfaces. Use higher-level domain subsystem interaction diagrams to develop the lower-level domain subsystem interaction diagrams. This activity is recursive.
- With UCs, DMs, and the domain subsystem interaction diagrams, develop the low-level object interaction diagrams. The main goal of this step is to capture:
  - how objects collaborate to accomplish the functionality described in UCs,

  - definitions of object interfaces,
  - object associations and interactions, and
  - the timing of the objects' interactions.

- With all interaction diagrams, build a unified collaboration diagram (UCD) without message numbering, multiple objects of the same type, or object names. Indicate:
  - *Controller* and *coordinator objects* — the main sources for the definitions of *active objects* in the design phase.
  - *Entity* and *service objects* — the main sources for the definitions of *passive objects* in the design phase.
  - *External objects* — the main sources for the definitions of *interfaces* in the design phase. These objects include devices and business resources.
- With the UCD, record each message as a method in the DM.
- For each controller and coordinator object in the UCD, develop a state diagram. The messages from the UCD are the main sources of events; the mapping is not necessarily one to one.
- Develop state diagrams for any additional objects that have non-trivial state transitions. The messages from the UCD are the main sources of events; the mapping is not necessarily one to one due to the possible presence of internal events.

Additional Notes: It is usually recommended to capture invariants, pre-conditions, and post-conditions for each entity in the OOA artifacts. Each entity or artifact has to be taken into account and to be related its constraints, business goals, business rules, non-functional requirements, and other artifacts captured during the requirements modeling phase.

6.3.3 OOD

Input: All requirements and domain model artifacts, with special emphasis on the DM, the UCD, and the state diagrams.

Goal: Map the domain model into an OOD model taking into account internal CBS requirements and development resources.

Activities: The main OOD activities are:

- Using the domain subsystem information from the DM and internal architectural requirements, design the initial high-level non-run-time architecture of the CBS. Define the interfaces of the CBS and its subsystem.
- Using the DM and the UCD, in addition to internal CBS requirements, map domain concepts into software classes. This mapping should be performed taking into account reusability, maintainability and other design goals. Take into account the internal CBS requirements such as persistence, security, performance, and so on. Augment and refine the

class interfaces.

- Define the run-time architecture of the CBS. Define run-time components, processes, and processing node allocations.
- Define the run-time communication channels, interfaces, and protocols.
- For each run-time entity, i.e., component, process, or communication channel:
  - make its decomposition explicit, i.e., define out of which objects it is constructed, and
  - make a clear distinction between active objects, i.e., controllers and coordinators, and passive objects, i.e., data concepts, computation and logic providers.
- Refine all class interfaces.
- For each class, design its internals, i.e., its algorithms, additional classes, data types, internal attributes, and so on.

There are many different OOAD methods, but common to all of them is an early decomposition of the domain into concepts. These concepts drive specification and have an impact on all the produced OOAD artifacts and, in some cases, propagate all the way to the code. This propagation of the concepts and the concepts' influence on the other produced artifacts might be both positive and negative. However, this issue is beyond the scope of this thesis.

6.4 Requirements Specification Techniques

Prior to discussing different specification techniques, it is important to emphasize different CBS aspects from a RE perspective. The four main aspects of each CBS from the RE perspective are processes, data, architecture, and interfaces. Each of most requirements specification techniques focuses on modeling one of these four main aspects. Nevertheless, in many an article in the requirements literature, this division is represented slightly differently. Typically, a discussion of a requirements specification techniques, requires discussing four orthogonal aspects of the domain and CBS that have to be modeled:

- *functional decomposition* — functions that are performed at the different abstraction levels,
- *behavior* — functions and control linked through temporal relationships,
- *communication* — spatial relationships among the different elements, and
- *conceptual analysis* — relationships among elements at the different abstraction levels.

The orthogonality is visible through the linkage of:

- *processes* and *interfaces* with the *functional decomposition* and *behavior*, and
- *data* and *architecture* with the *conceptual analysis* and *communication*.

A typical further division is of the *CBS-level* and *internal* aspects. *System-level* aspects consider the CBS as a black box, and thus we have *CBS functions*, *CBS behavior*, *CBS communication* with the CBS's environment, and *CBS conceptual analysis* with respect to CBS's environment. *Internal* aspects capture requirements-related properties of the internal components of the CBS. This level usually shows *component functions*, *component behavior*, *component communication*, and an *internal conceptual analysis*. This division of the *CBS-level* and of the *internal* aspects, although common, is not particularly relevant to this discussion as the same techniques are typically used to specify both *CBS-level* and *internal* aspects of the CBS.

Overall, we have four different sets of techniques to capture different CBS aspects:

- *functional specification techniques* — This category of techniques includes many different specification techniques [63], most of which yield declarative and imperative specifications, with probably the most popular ones being those involving UCs [22].

- *behavioral specification techniques* — This category of techniques includes *process graphs*, *Jackson's Structured Development Process Structure Diagrams*, *extended finite state diagrams*, *Mealy Machines*, *Moore Machines*, *SDL state diagrams*, and statecharts [98], with probably the most popular being *statecharts* [43]. Statecharts have been especially attractive because, from the beginning, they had a formal definition [43, 44]. Nevertheless,

by as early as 1994, there were over 20 different formal definitions of statecharts [96]. Statecharts have been successfully used for both structured requirements specification [45] and OO requirements specification [46, 62].

- *communicational specification techniques* — This category of techniques includes *dataflow diagrams (DFD)*, *context diagrams*, *SADT activity diagrams*, *StateMate activity charts*, *object communication diagrams*, *JSD system network diagrams*, *UC diagrams*, *SDL block diagrams*, *sequence diagrams*, and *collaboration diagrams* [98]. Since the standardization of UML, *sequence diagrams* and *collaboration diagrams* are probably the most widespread communication specification techniques.

- *conceptual analysis techniques* — This category of techniques includes *entity-relationship diagrams* and *class diagrams* [98]. Each is in widespread use, the former in database and information systems development, and latter in most OO methods.

It is important to note that each of most of these techniques does not belong only to one category. For example, although the main purpose of the *UCs* is to capture *functional decomposition*, *UCs* capture also certain *behavioral*, *communicational*, and *conceptual analysis* aspects.

In the SRSs of the case studies described in this thesis work, there was an extensive use of *UCs*, *UC diagrams*, *sequence diagrams*, *statecharts*, and *class diagrams*. Together they covered all four main aspects of a CBS's domain model that had to be captured for a complete requirements specification. Moreover, they

are all recommended by most of the current OOAD methods.

6.5 Goal-Driven Requirements Engineering

As mentioned in this chapter's introduction, several researchers began studying goal-driven RE [24, 72, 94, 10] as a promising technique for dealing with domain-level requirements for large CBSs. Goal-driven RE focuses on ensuring that software actually fulfills business needs and requirements. This focus has been achieved by shifting from considering *what* a CBS should do to considering *why* the CBS should provide particular functionality. In other words, *requirements rationale* is the main focus.

Although each of most of the original goal-driven RE techniques concerns domain-level requirements, for example, through analysis of *personal* and *CBS* goals, the main idea of goal-driven RE techniques has been used to enhance certain traditional requirements techniques such as UCs [22]. Nevertheless, although goal-driven RE techniques are extensively studied, goal-driven RE remains an immature area. This immaturity is apparent from the many different definitions of the word "goal" [24, 7, 72]. From the author's perspective, goals capture the *intention, i.e., objective,* and the *target state* for the entity under analysis and at the entity's own abstraction level. For example, in the case of an elevator CBS, a goal for the elevator CBS is to *deliver passengers to the requested floor*. This goal captures the *intention* of *delivering passengers* and also the *target state* of *arriving at the requested floor*. This particular goal captures the rationale for the elevator's

responsibility for carrying passengers from a floor to another.

An interesting point to note is that depending on the abstraction level from which one is observing a CBS and the goal decomposition, a goal may or may not become a functional requirement. For example, for an elevator CBS, the next level of the goal decomposition might include goals such as *move elevator cab*, *stop elevator cab*, *pick up a passenger*, and so on. Now, if we start working at this abstraction level, the higher-level goal of *delivering passengers to the requested floor* becomes a functional requirement for the lower-level goals such as *moving elevator cab*. An advantage of this goal hierarchy is that it provides traceability when moving from one abstraction level to another and from one goal decomposition level to another.

For the purposes of OOA work, it is important to carry the goal idea to the product-level requirements specification. In particular, matching of a concept's goals and the concept's method goals is important for building meaningful CBSs. It is important to avoid a mismatch in concept's goals and its methods. A mismatch contributes to the OOA model's inconsistencies.

Contrary to other goal-driven RE methods, in OOA work, one takes advantage of analyzing goals only indirectly. For example, one does not typically attempt to build different goal-based artifacts such as goal decomposition trees or goal-conflict tables. Instead, we use goals only for low-level verification of the purposes of concepts; most often just intuitively. We also do not typically attempt to take advantage of other related goal-concepts such as *soft goals*, *obstacles*, and *quality attributes* in OOA, all of which might significantly improve OOA.

6.6 Unified UC Statecharts

This section compares the work of this thesis to that of the three papers whose work appears to be closest, namely papers by Glinz, Whittle *et al.*, and Harel *et al.* [39, 97, 47]. Each of these papers describes one formal treatment of unification of UCs into a statechart similar in semantics to our unified UC statechart[2]. Several others, including Somé *et al.* [88], van Lamsweerde *et al.* [95], and Khriss *et al.* [55] have describe algorithms and methods for synthesizing various domain models, including one in statechart notation, from UCs.

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

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

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. 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.

There are many analogies between the steps, restrictions, and problems in the methods and algorithms of Glinz, Whittle *et al.*, and Harel *et al.* and those of UCUM, 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 that were observed of UCUM. Thus, it can be said that this thesis work and their work constitute independent confirmations of each other.

CS N3TS, CS N12ES, and CS N46VS 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. CS N3TS, CS N12ES, and CS N46VS have shown UCUM to provide specific practical benefits to the analysts who apply it and have exposed the drawbacks of the method. A case study of an actual method use can measure the cost of applying the method. In

particular, CS N3TS, CS N12ES, and CS N46VS 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%. Because the analysts in CS N3TS, CS N12ES, and CS N46VS were students with no expertise in either statecharts 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 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.

Chapter 7

Conclusion

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, each with none to several years of software development experience. The average number of concepts in the SRSs of CS N34VS was 5.5% higher than the average number of concepts in the SRSs of CS O31VS. The average number of common concepts in the SRSs of CS N34VS was 14.4% higher than the average number of common concepts in the SRSs of CS O31VS. The concept concentration in the SRSs of CS N34VS was higher than in the SRSs of CS O31VS for all and the common concepts. Therefore, the semantic similarity in the captured concepts from one SRS to another was higher in the SRSs of CS N34VS than in the SRSs of CS O31VS. So, the author made a conservative estimate that the semantic similarity of concepts is **approximately 10% higher**

in the SRSs of CS N34VS than in the SRSs of CS O31VS, based on

- an average increase of 5.5% in the number of concepts per SRS of CS N34VS over the number of concepts per SRS of CS O31VS
- an average increase of 14.4% in the number of common concepts per SRS of CS N34VS over the number of common concepts per SRS of CS O31VS
- an increase in the capture of common concepts per SRS from 75% in the SRSs of CS O31VS to 84% in the SRSs of CS N34VS;
- a narrower data spread for both sets of data; and
- an overall qualitative analysis.

The semantic similarity of the SRSs of CS N34VS was higher than the semantic similarity of the SRSs in CS O31VS. However, the cost of producing the SRSs in the CS N34VS was measurably higher than the cost of producing the SRSs in CS O31VS. The cost of producing an SRS was approximately 25% higher for the SRSs of CS N34VS than for the SRSs of CS O31VS.

This evaluation led us to a conclusion that:

> **SDP can be reduced by approximately 10% by performing a detailed behavioral analysis before conceptual analysis.**

This reduction was at the cost of approximately 25% increase in the analysis workload. The two numbers, 10% and 25%, are not comparable and should not be used to estimate increased semantic similarity and UCUM's impact on the

overall development process and the CBS being built without taking additional model and process measurements into account. There is no easy way to estimate the benefits caused by the 10% improvement in the semantic similarity to the downstream development speed, reliability, robustness, and other qualities of the CBS being developed. Also, there is no easy way to estimate benefits UCUM brings in downstream development in terms of improved reliability, robustness and other qualities for the specified CBS.

Overall, these case studies exposed the following impact of the use of UCUM and extensive functional and behavioral modeling before proceeding with conceptual analysis:

1. better functional analysis and discovery of more requirements,
2. an increase in the semantic similarity patterns,
3. several qualitative changes:
    - larger number of *functional* concepts, i.e., *processors*,
    - clearer definition of *interface* concepts, and
    - lack of *inheritance*, and
4. still insufficient number of *data* concepts.

The detailed functional and behavioral modeling using unified UC statecharts is very similar to traditional structured analysis [e.g., 27, 100, 99, 98]. So, combining structured analysis and OOA helped improve the semantic similarity and the quality of DMs in a way summarized in [81]:

> A weakness of OO is that OO methods only build functional models within the objects. There is no place in the methodology to build a complete functional model. While this is not a problem for some applications (e.g., building a software toolset), for large systems, it can lead to missed requirements. Use cases address this problem, but since all use cases cannot be developed, it is still possible to miss requirements until late in the development cycle.

In addition, it is probably this integration of the detailed functional and behavioral analysis with OOA that has resulted in the deeper, more detailed, and "closer to design and architecture" DMs, as observed by independent reviewers [e.g., 69] and some graduate students familiar with other OOA methods who did SRSs of CS N12ES [1]. The same reviewers observed that the method itself is more systematic than many scenario-based OOA methods. The DMs produced using UCUM were felt to provide a more solid base for transitioning to design phases than those produced by the OOA methods that had been used previously.

7.1 Contributions

The main contributions of this thesis are:

- Analysis and interpretation of data that helps us further our understanding and knowledge about OOA methods and the results they produce.
- Evaluation of the impact that traditional functional analysis brings to the

OOA and the relationship between them.

- Specification of an OOA method based on unified UC statecharts.
- Analysis of the semantic similarity of independently specified DMs as the means for evaluating predictability of the results of OOA methods used to produce these DMs.
- Offering and using the idea of the semantic similarity measure as the means for evaluating predictability of the results of OOA methods.

7.2 Limitations

The main limitations of this research are:

- Not everyone agrees that independently specified conceptual models of the same domain system should be semantically similar.
- The evaluation has taken into account only conceptual models as recorded in the SRSs. No analyst was asked for an explanation of what his or her models mean.
- Semantic similarity of conceptual models was evaluated only through a comparison of concepts. Future work should take into account other parts of the conceptual models, such as relationships and stereotypes.

- Only small to medium sized reactive, real-time, and information domain system conceptual models were evaluated. No conceptual models of very large transformational domain system were evaluated.

7.3 Future Work

The limitations of the case studies presented in this thesis and of the conclusions suggest future work:

- Independent replication of the case studies.
- Controlled experiments.
- Use of different measures of model quality.
- Studies of OOA performed on larger domain systems.
- Studies of other variations of the methods, such as usage of activity diagrams, different use case formats, different notations for conceptual models, goal-based and aspect-oriented extensions to OOA methods, etc.
- Examination of the design and implementation artifacts produced by the same students in response to the SRSs that were examined in the case studies of this thesis.

Appendix A

Elevator System Case Study: CS O3ES

This appendix presents the main results of the case study CS O3ES, which was reported in a paper published by this author and his advisors [90]. Except for some minor changes, the text of this appendix is unchanged from the paper. This case study was conducted in order to ascertain if the problems observed in students' projects also exist in much smaller domain specifications.

A.1 Case Study Hypothesis

The current trend of software development is towards iterative and use-case driven [82, 62] processes. In such processes, most domain objects are discovered iteratively, and the main source for their discovery are UCs and the domain knowledge

acquired during development of the UCs. The students' projects were conducted in this manner. Depending on the abstraction level of the UCs, the degree to which sub UCs are separated out, and on how many scenarios are abstracted into a UC, a typical SRS had anywhere between 10 and 30 UCs.

The difficulty of discovering domain concepts does not appear to be greatly affected by the overall size of the domain, because the conceptual decomposition was done at the level of the UCs. Since the conceptual decomposition was use-case driven, i.e., concepts were discovered as they came up during the generation of UCs, we have come to believe that the *CID of OOA is mostly independent of the size of the domain under consideration*. In order validate the correctness of this assumption, we have decided to perform a comparative case study of three specifications of a much smaller domain: an elevator domain. The discussion about these specifications serves also to illustrate concretely the difficulties of object-oriented concept decomposition.

The hypothesis explored in this case study is that

the CID is present in both small and large domains,

i.e., the difficulties that we have observed in students' work are due not to the size of the domain they were specifying but rather to something else, perhaps directly related to the object-oriented analysis paradigm.

A.2 Research Method

In order to test our hypothesis, we decided to perform a comparative study of several independently produced specifications of elevator domains. We chose three different specifications found using Internet search engines.

Each of the specifications specifies the basic functionality of the elevator as seen from a user's perspective. This view of the elevator domain means that there are two basic high-level UCs considered:

1. UC1: request an elevator cab to move to a particular floor, from *outside* the elevator cab, and

2. UC2: request an elevator cab to move to a particular floor, from *inside* the elevator cab.

The number of UCs in an elevator domain is approximately one tenth of that of the telephone domain with which the students were dealing. At the same time, the elevator domain is of non-trivial size, as it comprises about 40 domain concepts.

A.2.1 Choice of the Case Study

To choose the case-study subject SRSs, we were guided by several requirements and constraints:

1. The hypothesis that the CID is independent of the size of a domain required us to look for a domain which is considerably smaller than that of the telephone domain used in the students' projects. The elevator domain satisfies

this criterion. It is also advantageous that many specifications of it, in a variety of sizes and degrees of completeness, are readily available on the Internet. The elevator domain has been used as an exemplar for years to demonstrate specification languages and techniques.

2. The chosen specifications were published on the Internet with no restrictions on their use for research purposes.

3. Each specification was authored by people with formal computer science education.

4. If we choose elevator domain specifications that were composed as pedagogical examples to show the strengths of object-oriented analysis and design, we expect fewer instances of the CID in the chosen specifications.

5. The elevator domain is familiar enough to most readers, allowing the discussion here to focus on modeling difficulties rather than on the details of the domain.

6. The focus of each specification we found is different; some are general domain modeling exercises, some are specifications of elevator management systems, and some are for simulation purposes. We decided to use three with different foci for a more robust test of the hypothesis. However, we expected that, nevertheless, their complete analysis models would be similar.

7. An elevator domain should be easier to analyze than most business domains, as the services, i.e., functionality, that an elevator offers are quite

simple, and the domain itself consists mostly of tangible, physical objects. In contrast, the typical business domain provides many complex interrelated services, and consists of many abstract, conceptual entities.

A.3 Analysis

This section first introduces all three SRSs and then presents the results of our analysis.

The viewpoint we took in this analysis is that of an *ignoramus* [8]; we intentionally did not attempt to learn the domain or specify an elevator domain ourselves before attempting this analysis. Also, we assumed each specification to be correct until it was proved otherwise. Finally, we assumed that object-oriented analysis is ideal for elevator domains, and we did not attempt any other kind of analysis. This viewpoint and these assumptions were required in order to preserve our objectivity in the case study.

A.3.1 First SRS

The first SRS [29] has the smallest specification of the three. Its main purpose is to teach the basics of UML. Its author focused on analyzing the basic elevator functionality from a passenger's perspective.

The published analysis consists of

1. a problem description,

2. a use-case diagram,

3. a description of each UC's basic scenario,

4. a collection of sequence and collaboration diagrams, one of each for each UC's basic scenario, and

5. a conceptual diagram.

The author does not provide full use-case descriptions. Since a problem description was provided, and it was used as the main source for the concept decomposition, the lack of full use-case descriptions is not a concern.

The author does not make any attempt to distinguish among the types of concepts in the conceptual diagram, and he does not clearly demarcate the domain's boundary. We suspect that not distinguishing among the types of concepts and not defining the boundary impeded his efforts to discover domain concepts. Nevertheless, we believe that this impediment had less of an impact in this SRS than in the course projects due to the smaller size of the case-study domain.

A.3.2 Second SRS

The second SRS [30] specifies an elevator control system for a three-story building. The size of this SRS is similar to that of the first SRS.

The published analysis consists of

1. a problem description,

2. a use-case diagram,

3. a conceptual diagram, and

4. a collection of state machine diagrams, one for each object in the conceptual diagram.

As in the first SRS, the author provides no complete UC description. The specification is based on concept extraction from the problem description.

A helpful feature in this domain specification is the names of discovered domain concepts are bold faced in the problem description. What is bold faced is a good indication that the author used a noun-extraction technique to identify the domain concepts.

As in the first SRS, the author does not make any attempt to distinguish among different types of concepts in the conceptual diagram, and he does not clearly demarcate the domain boundary.

A.3.3 Third SRS

The third SRS [31] specifies a system for control of multiple elevators in a high-rise building. The system is supposed to be able to support from one to eight elevators, the exact number being a parameter of the specification. Each building has its own number of floors. Each elevator serves a possibly different set of non-adjacent floors; the set of floors served by an elevator is called a *part* of the building. Each part of the building will have no more than four elevators installed to serve it.

The published analysis consists of

1. a problem description,
2. a use-case diagram,
3. use-case descriptions, one for each UC, and
4. a specification in the form of a collection of state machine diagrams, one for each object mentioned in the problem description and the UC descriptions.

Unlike in the first two SRSs, this SRS has fully developed use-case descriptions in addition to a problem description that is about the same size as the problem descriptions of the first two SRSs. This SRS's problem description is focused on the domain's structure rather than on the domain's functionality and constraints.

The main component of interest for us, the conceptual diagram, is not provided. Instead, the specification is divided into different sections, one for each concept; and for each concept, an extensive set of state diagrams is given.

A.3.4 Comparison

We performed an analysis of each of these SRSs to find all concepts present in any SRS. Table 1 shows the union of all concepts found in the SRSs; the unification was performed on the bases of (1) the names assigned to the concepts, i.e., the same name appearing in two SRSs is assumed to name the same concept in both SRSs, and (2) the meanings of concepts, i.e., two concepts in different SRSs that mean the same thing are considered to be the same concept. Each concept has a row in the table. For each concept and each SRS, the intersection of the concept's

row and the SRS's column has an entry indicating the origin of the concept within the SRS:

- "D" indicates that the concept was discovered by the study's author.
- "I" indicates that the concept was not discovered or was ignored by the study's author, although noun extraction shows that the concept is clearly in the domain, and
- an empty slot indicates that it was not possible to discover the concept from any domain artifact mentioned in the SRS.

A concept labeled "D" is called a "D concept", and a concept labeled "I" is called an "I concept".

The table contains also a *type* column indicating the types for its intersecting concepts. Concept types help in classifying and understanding concepts. The concept types used in the study are:

- Physical Structural Element (PSE): A PSE is an entity that has a responsibility to act as a physical boundary, container, or structural element in a physical system.
- Conceptual Structural Element (CSE): A CSE is an abstract entity that has a responsibility to act as a concept boundary, container, or structural element primarily in an abstract domain, e.g., a department in a company is a CSE.
- Physical Processor (PP): A PP is a physical device that performs computations within the domain.

- Conceptual Processor (CP): A CP is an abstract entity that performs computations within the domain.
- Actor (A): An A is an external entity that directly communicates with the domain.
- Intangible Concept (IC): An IC[1] is an abstract entity that exists within the domain.

The concept rows are sorted by the concepts' types.

Physical structural elements and *actors* play important roles in the definition of the domain boundary and interface. In our experience, these two types of concepts are the easiest ones to discover in the domain. *Physical processors* are important for the domain's interface definition. They are relatively easy to discover, but often difficult to decompose into components. *Conceptual structural elements*, *conceptual processors*, and *intangible concepts* are important for the internal design of the CBS. These concepts are the most difficult to discover primarily due to their abstract nature and their only implicit existence within the domain.

Finally, the table contains a *purpose* column indicating the purposes that concepts take on in the specifications in which they were indicated.

A total of 44 concepts were discovered in the three specifications. Table A.2 shows the numbers of D and I concepts in the three SRS columns of Table 1. It shows also for each number of D or I concepts, its percentage out of all concepts

[1]We use the "*Intangible* Concept" instead of just "Concept" as the name of the type of an abstract entity to avoid confusion with general term "concept" used to describe arbitrary concepts that appear in the model.

Concept	*SRS1 Status*	*SRS2 Status*	*SRS3 Status*	*Type*	*Purpose*
elevator system	I	D	I	CSE	to define the conceptual boundaries of the system to control and move the elevator cab
passenger	I	I	I	A	to use the elevator
elevator cab	D	D	D	PSE	transport passengers
building			I	PSE	physical system boundary
floor	I	D	I	PSE	to provide building's structure to define elevator travel destinations
top floor		I	I	PSE	special floor - different user interface to provide direction reference
bottom floor		I	I	PSE	special floor - different user interface to provide direction reference
button panel			I	PSE	container for buttons
elevator shaft			I	PSE	pathway for the elevator cab provide elevator access to the floors
button	D		I	PP	to unify all buttons
elevator button	D	D		PP	unify buttons inside the elevator cab
floor request button		D	D	PP	user interface for requesting floors
door button			D	PP	user interface for door opening
open door button		D	I	PP	request to open the doors when the elevator is not moving
close door button		D	I	PP	request to close the doors when the doors are open
floor button	D	D	D	PP	user interface for requesting elevator
stop button			I	PP	request immediate elevator stop at the next floor
door	D	D	D	PP	close the elevator cab
inner door		D		PP	close the elevator cab
outer door		D	I	PP	close the floor access to the elevator shaft
door opening device		I		PP	open the doors
floor number display			I	PP	user interface for indicating travel progress
floor sensor		D		PP	detect elevator position with respect to the floors
elevator engine		D		PP	move the elevator cab
elevator controller	D			CP	to delegate interface requests within the system to delegate internal responsibilities within the system
door timer		D	I	CP	constrain door opening time periods
current floor	I			IC	to define current location of the elevator
designated floor			I	IC	final travel destination
request		I	I	IC	unify all the request types
requested direction		D		IC	track user's traveling preference used for the elevator stopping scheduling purposes
elevator request		D		IC	track user's request for elevator services used for the elevator stopping scheduling purposes
elevator-up request		D		IC	*same as for elevator request*
elevator-down request		D		IC	*same as for elevator request*
pending queue		D		IC	keep track of unprocessed *elevator request button* and *floor request button* requests
summon			D	IC	to capture elevator request
stop request			I	IC	capture users request for stopping at a particular floor
time period		I	I	IC	constraint time allowed for various operations
stop			I	IC	unify different elevator stopping situations
immediate stop			I	IC	unplanned stop initiated by the passenger
planned stop			I	IC	stop at final travel destination
stop notification			I	IC	user interface for indicating elevator stops
button refusal notification			I	IC	user interface for invalid request notification
direction			I	IC	capture current traveling direction of the elevator
light		I		IC	user interface to indicate button status

Table A.1: All Discovered Concepts

Case Study	*Discovered*	*% Discovered*	*Ignored*	*% Ignored*
SRS1	6	13.64	4	9.09
SRS2	19	43.18	7	15.09
SRS3	6	13.64	25	56.82

Table A.2: Numbers of Concepts

found in any SRS.

In the first SRS, the ratio of discovered to ignored concepts is 3:2, in the second, the ratio is 2.7:1, and in the third, the ratio is 1:4.2. Clearly the ratios vary widely over the SRSs with no particular pattern. This observation is consistent with our experiences with the course projects, for which we could never predict how many concepts a particular team would manage to capture.

The concept type with the lowest D-to-I ratio is IC; probably because people generally have difficulty identifying intangible, abstract concepts. The type with the second lowest D-to-I ratio is PSE. We surmise that the authors perceived physical structural entities as being less important than other concepts of other types, because physical structural entities are perceived as being outside the scope of the domain. Nevertheless, these concepts should be captured because they often constrain the behavior of the internal domain.

A.4 Evaluation

This section's subsections contain evaluations of one SRS relative to three specific manifestations of the CID:

1. *Misplaced Responsibilities*: determining which "D" concepts were assigned

the responsibilities that really belong to the missing "I" concepts,

2. *Omitted Responsibilities*: determining which responsibilities mentioned in a problem description were entirely missed in the corresponding models, and

3. *Omitted Passive Concepts*: determining which concepts, either "D" or "I", are consumed or produced through interactions of other concepts.

Due to space limitations, mostly only the evaluation of the second SRS is presented. Because of the focus on one SRS, unless otherwise explicitly stated each "author" means the second SRS's author, and each published analysis artifact, e.g., the conceptual diagram, is that of the second SRS.

A.4.1 Misplaced Responsibilities

From our course project experience and the observations of these SRSs, it appears that emphasizing structure over function in decomposing a domain leads to difficulties assigning responsibilities to concepts. Moreover, when responsibilities are not clearly observable in a domain description, many activities remain unidentified.

Table A.3 shows, for each D concept that appears in the conceptual diagram, the activities assigned by the author to that concept. The table shows also the purposes of these concepts as derived by us from all three SRSs. Five of the eight D concepts in the conceptual diagram do not even have clear definitions of the activities for which they are responsible.

Concept	*Activity*	*Purpose*
elevator (cab)	*none*	transport passengers
elevator engine	*none*	move the elevator cab
floor button	request the elevator to come to the floor	user interface for requesting elevator
elevator button	*none*	unify buttons inside the elevator cab
open door button	request to open the doors when the elevator is not moving	user interface for opening door
close door button	request to close the doors immediately	user interface for closing door
door	*none*	close elevator cab and shaft access for the safety purposes
door timer	*none*	constrain door opening time periods

Table A.3: Second Case Study: Discovered Concept-Activity-Purpose Relationships

The main symptom of misplaced responsibilities is the existence of many I concepts in a conceptual diagram. When concepts are missing, an activity that is needed to fulfill the domain's functionality gets assigned to one of the D concepts, often to a not fully appropriate concept; the overloaded concept gets this additional activity in addition to the activities for which it should be responsible. This misallocation of responsibilities means that each D concept has to fulfill a number of responsibilities that really should be fulfilled by other concepts, often not present in the conceptual diagram.

Even for the three D concepts that have their activities clearly indicated, (1) floor button, (2) open door button, and (3) close door button, we can observe misplaced responsibilities. For each concept, the purpose field indicates responsibility for only a subset of the activities that have been assigned to the concept. According to the author, each of these concepts is responsible for *requesting the elevator to perform a particular activity*. However, the purpose of each of these concepts is to serve as a user interface for the corresponding request. Capturing a user request

is only a partial responsibility of the overall activity of requesting an elevator to perform an activity.

The reader may wonder why these three concepts cannot themselves completely fulfill the responsibility of requesting the elevator to do a particular activity. It is sufficient to identify the I concept that *should* collaborate with these three concepts to fulfill the responsibility. That one I concept would be request. This concept's purpose is to capture any request and all of its parameters and to carry out the actual request by distributing parameters to the concepts that participate in doing the request. This mode of thinking is important, because just discovering the request concept leads to the discovering a request's parameters. This analysis propagation is necessary to achieve a complete model.

A.4.2 Omitted Responsibilities

We assume that the author *was* able to identify responsibilities that were mentioned in the problem description but were omitted from the conceptual diagram; after all, the author wrote the problem description! Therefore, this subsection focuses on only I concepts that were neither indicated in the domain description (by the author's having used bold face in the problem description) nor included in the conceptual diagram.

The first I concept to consider is time period. The concept time period is used in the activity of constraining the amount of the time the elevator door is open. The concept that directly depends on and uses time period is door timer. Since time period is not explicitly captured as a concept, and since the door timer concept

does not capture the notion of having to keep track of the amount of time for which the door can stay open, the responsibility of keeping track of time is omitted.

The second omitted responsibility is that of opening doors. This responsibility should be be assigned to door opening device. It is possible that the author assumed that this activity is part of a door's functionality. However, because this activity is captured neither in the domain description nor in the diagrams, we assume that it was missed entirely or purposely omitted. In addition, that this responsibility must exist is clear from the existence of the open door buttons and the close door buttons. Obviously, the author had discovered two out of three concepts that participate in the activity of managing door movement but omitted the concept that would have been responsible for the actual action of moving the doors.

The light concept's responsibility to indicate a button's status is missing. This responsibility might have been identified but purposely omitted if the author assumed that responsibility is handled by the button's hardware and thus does not need to be in the software. However, even when hardware does discharge a responsibility, the responsibility needs to be specified so that the responsibility is not lost if hardware that behaves differently is ever used in the future.

The author used elevator request button in the conceptual diagram to unify the buttons and button requests. However, we believe that request should have been a concept in its own right in order to unify all elevator requests. Thus, request is considered to be a partially omitted responsibility.

Another group of obvious, but omitted, responsibilities is those of the pas-

senger, as the user of the elevator. Since some argue that actors should not be included in conceptual diagrams, it is possible that the author made an explicit decision to omit the passenger's responsibilities.

Finally, the unique responsibilities of the top floor and the bottom floor are to deal with the different user interfaces that these floors require. Also, the responsibility of these two floors to provide a direction reference for the elevator cab has been omitted.

A.4.3 Omitted Passive Concepts

The specification has a rich set of passive concepts. The passive concepts that the author has identified in the domain descriptions are requested direction, elevator request, elevator-up request, and elevator-down request. Although these passive concepts were clearly indicated in the domain descriptions, the author did not include any of them in the conceptual diagram, probably because of their passive nature. None of them is performing active work. Instead they are produced or consumed by other concepts in achieving the other concept's responsibilities. This omission is consistent with what we have seen in students' projects.

Note that the request concept is an abstract concept rather than a passive one, since its purpose is to unify many concrete passive concepts. When considering the students' projects, we observed that abstract concepts need to be discovered because their discovery often facilitates the discovery of other passive concepts. This facilitation could be regarded as one purpose of identifying inheritance during analysis. In the SRS, however, the author did manage to discover several

related concrete passive concepts without discovering this abstract concept.

The third SRS is quite similar to the second with respect to the discovery of passive concepts. The third SRS's author discovered and included only one passive concept: summon. We discovered several additional ones: notification, direction, stop, time period, request, stop request, and stop notification. Overall, in all three SRSs, passive concepts were largely omitted, whether from ignorance or inability to discover them.

Appendix B

CS N3TS — Remaining Diagrams

This appendix shows other diagrams that were produced during the Turnstile System OOA.

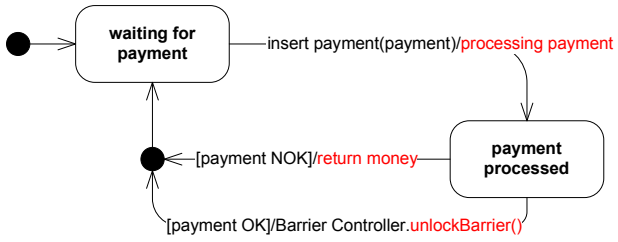


Figure B.1: State diagram for Payment Processor

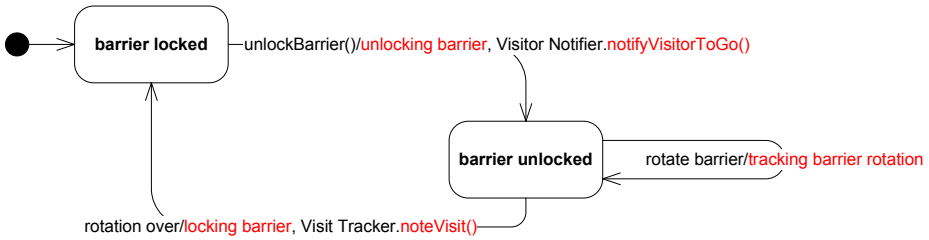


Figure B.2: State diagram for Barrier Controller

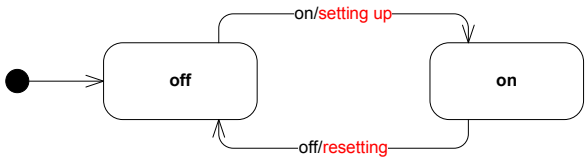


Figure B.3: State Diagram for System Status Controller

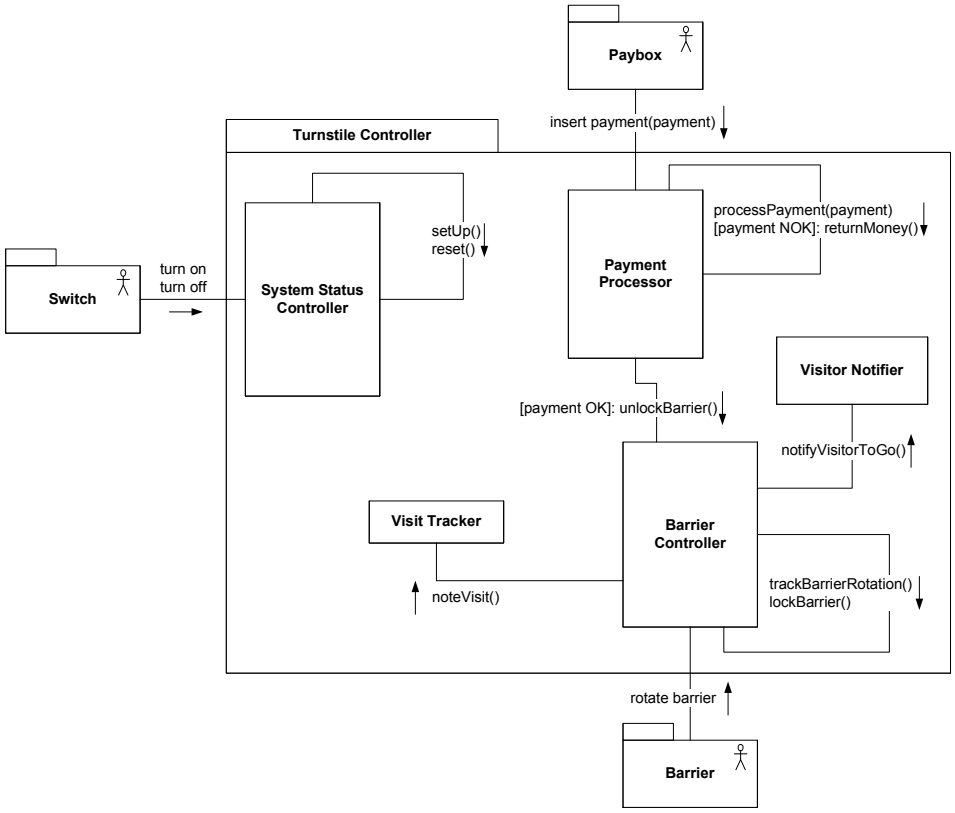


Figure B.4: Unified System Collaboration Diagram

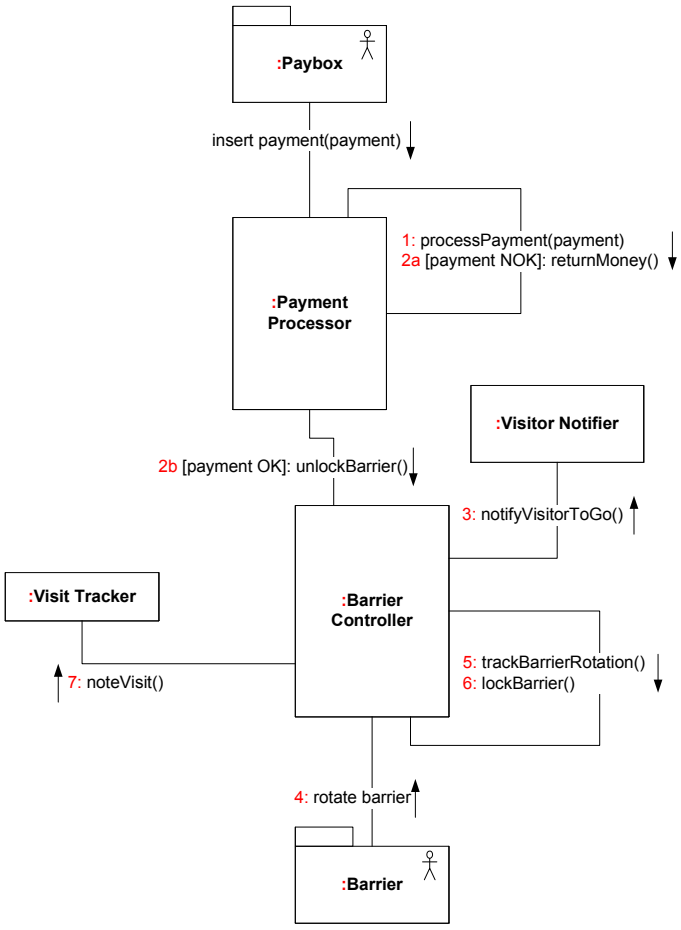


Figure B.5: UC3 Collaboration Diagram

Appendix C

Abbreviations

CBS Computer-Based System

CID Concept Identification Difficulty

DM Domain Model

IMS Information Management System

OO Object-Oriented

OOA Object-Oriented Analysis

OOAD Object-Oriented Analysis and Design

OOD Object-Oriented Design

OOP Object-Oriented Programming

RE Requirements Engineering

SDL Specification and Description Language

SDP Semantic Dissimilarity Problem

SRS Software Requirements Specification

TA Teaching Assistant

UC Use Case

UCUM Use Case Unification Method

UML Unified Modeling Language

VoIP Voice-over-IP

Bibliography

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

[2] How elevators work. http://science.howstuffworks.com/elevator.htm; accessed June 10, 2006.

[3] Semantic similarity definition. http://en.wikipedia.org/wiki/Semantic_similarity; accessed September 15, 2006.

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

[5] Russell Abbott. Program design by informal english descriptions. *Communications of the ACM*, 26(11), November 1983.

[6] Ian Alexander. RE'05. *Requirements Quarterly*, RQ37:6–9, September 2005.

[7] Annie I. Antón. Goal-based requirements analysis. In *ICRE '96: Proceedings of the 2nd International Conference on Requirements Engineer-*

ing (ICRE '96), page 136, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7252-8.

[8] Daniel M. Berry. The importance of ignorance in requirements engineering. *Journal of Systems and Software*, 28(2):179–184, 1995. ISSN 0164-1212.

[9] Graham M. Birtwistle, Ole-Johann Dahl, Bjørn Myhrhaug, and Kristen Nygaard. *Simula Begin*. Studentlitteratur, Lund, Sweden, 1980.

[10] Barry Boehm and Hoh In. Identifying quality-requirement conflicts. *IEEE Software*, 13(2):25–35, 1996.

[11] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, CA, USA, second edition, 1994.

[12] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, Reading, MA, second edition, 1994.

[13] Grady Booch. Object-oriented design. *Ada Lett.*, I(3):64–76, 1982. ISSN 1094-3641. doi: http://doi.acm.org/10.1145/989791.989795.

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

[15] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Modeling early requirements in tropos: A transformation

based approach. In *AOSE*, pages 151–168, 2001. URL `citeseer.nj.nec.com/443771.html`.

[16] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. A knowledge level software engineering methodology for agent oriented programming. In *AGENTS '01: Proceedings of the Fifth International Conference on Autonomous Agents*, pages 648–655. ACM Press, 2001.

[17] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Addison-Wesley, Boston, Massachusetts, first edition, 1995.

[18] Jaelson Castro, Manuel Kolp, and John Mylopoulos. A requirements-driven development methodology. *Lecture Notes in Computer Science*, 2068:108–??, 2001. URL `citeseer.nj.nec.com/castro01requirementsdriven.html`.

[19] Jaelson Castro, Manuel Kolp, and John Mylopoulos. Towards requirements-driven information systems engineering: The tropos project. *To Appear in Information Systems, Elsevier, Amsterdam, The Netherlands*, 2002. URL `citeseer.nj.nec.com/castro02towards.html`.

[20] Peter Coad and Edward Yourdon. *Object Oriented Analysis*. Prentice Hall, Englewood Cliffs, NJ, second edition, 1990.

[21] Alistair Cockburn. Private communication, 2006.

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

[23] Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *SIMULA 67 Common Base Language*. Norwegian Computing Centre, Oslo, Norway, 1968.

[24] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. In *6IWSSD: Selected Papers of the Sixth International Workshop on Software Specification and Design*, pages 3–50, Amsterdam, The Netherlands, The Netherlands, 1993. Elsevier Science Publishers B. V. doi: http://dx.doi.org/10.1016/0167-6423(93)90021-G.

[25] Thomas H. Davenport. *Process Innovation – Reengineering Work through Information Technology*. Harvard Business School Press, Boston, first edition, 1993.

[26] Alan M. Davis. *Software Requirements: Analysis and Specification*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1990.

[27] Tom DeMarco. *Structured Analysis and System Specification*. Yourdon Press, New York, 1978.

[28] 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.

[29] ES1. Elevator system specification. `http://www.geocities.`

com/SiliconValley/Network/1582/uml-example.htm; accessed January 10, 2006.

[30] ES2. Elevator system specification. http://se.uwaterloo.ca/~mctanuan/thesis/elevreqt.ps; accessed January 10, 2006.

[31] ES3. Elevator system specification. http://www.umot.net/examples/elevator.php; accessed January 10, 2006.

[32] Shamal Failey. Does object-oriented analysis work? *Requirements Quarterly*, RQ37:10–11, September 2005.

[33] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, Boston, Massachusetts, third edition, 2004.

[34] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. John Wiley & Son Ltd, Hoboken, N.J., first edition, 1996.

[35] Paolo Giorgini, Anna Perini, John Mylopoulos, Fausto Giunchiglia, and Paolo Bresciani. Agent-oriented software development: A case study. In *Proceedings of the Thirteenth International Conference on Software Engineering & Knowledge Engineering (SEKE01)*, 2001.

[36] Fausto Giunchiglia, John Mylopoulos, and Anna Perini. The tropos software development methodology. *Technical Report No. 0111-20, ITC - IRST. Submitted to AAMAS '02. A Knowledge Level*

Software Engineering 15, 2001. URL `citeseer.nj.nec.com/giunchiglia01tropos.html`.

[37] Fausto Giunchiglia, Anna Perini, and Fabrizio Sannicolo. Knowledge level software engineering. *In Springer Verlag, Editor, In Proceedings of ATAL 2001, Seattle, USA. Also IRST TR 011222, Istituto Trentino Di Cultura, Trento, Italy*, 2001. URL `citeseer.nj.nec.com/giunchiglia01knowledge.html`.

[38] 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.

[39] 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.

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

[41] 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.

[42] Michael Hammer and James Champy. *Reengineering the Corporation: a Manifesto for Business Revolution*. Nicholas Brealey P., London, first edition, 1995.

[43] 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.

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

[45] David Harel and Eran Gery. Executable object modeling with statecharts. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 246–257, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7246-3.

[46] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/235321.235322.

[47] 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, Jan 2005.

[48] Les Hatton. Does OO really match the way we think? *IEEE Software*, 15 (3):46–54, 1998.

[49] Ric Holt. Private communication, 2006.

[50] Daniel Ingalls. The smalltalk-76 programming system: Design and implementation. *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages*, January 1978.

[51] Michael A. Jackson. The role of architecture in requirements engineering. In *Proceedings of the IEEE International Conference on Requirements Engineering*, page 241. IEEE Computer Society, 1994.

[52] Frederick P. Brooks Jr. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[53] Judith Kabeli and Peretz Shoval. Data modeling or functional analysis: What comes next? an experimental comparison using FOOM methodology. In *Procedings of the Eighth CAISE–IFIP WG 8.1 International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design (EMMSAD'03)*, pages 48–57, 2003.

[54] Hermann Kaindl. Is object-oriented requirements engineering of interest? *Requirements Engineering*, 10(1):81–84, 2005.

[55] Ismal 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.

[56] Tomoji Kishi and Natsuko Noda. Aspect-oriented analysis for architectural design. In *IWPSE '01: Proceedings of the Fourth International Workshop on Principles of Software Evolution*, pages 126–129. ACM Press, 2001.

[57] Manuel Kolp, Paolo Giorgini, and John Mylopoulos. A goal-based organizational perspective on multi-agent architectures. In *ATAL '01: Revised Papers from the 8th International Workshop on Intelligent Agents VIII*, LNCS 2333, pages 128–140. Springer, 2002.

[58] Axel Korthaus. Using UML for business object based systems modeling. In Martin Schader and Axel Korthaus, editors, *The Unified Modeling Language – Technical Aspects and Applications*, pages 220–237, Heidelberg, Germany, 1998. Physica.

[59] Jeff Kramer. Abstraction: The key to software engineering? *Keynote: JSSST Japan Society for Software Science and Technology Conference*, 2004.

[60] John Krogstie, Odd Ivar Lindland, and Guttorm Sindre. Towards a deeper understanding of quality in requirements engineering. In JuhanI Iivari, Kalle Lyytinen, and Matti Rossi, editors, *Proceedings of Seventh International Conference on Advanced Information Systems Engineering (CAiSE'95)*, pages 82–95. Springer, 14–16 June 1995.

[61] John Krogstie, Guttorm Sindre, and Håvard Jørgensen. Process models

representing knowledge for action: A revised quality framework. *European Journal of Information Systems*, 15(1):91–102, February 2006.

[62] 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.

[63] Søren Lauesen. *Software Requirements: Styles and Techniques*. Addison-Wesley, Reading, MA, 2002.

[64] 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.

[65] Odd Ivar Lindland, Guttorm Sindre, and Arne Sølvberg. Understanding quality in conceptual modeling. *IEEE Software*, 11(2):42–49.

[66] Peter Loos and Peter Fettke. Towards an integration of business process modeling and object-oriented software development. Technical report, Chemnitz University of Technology, Chemnitz, Germany, date unknown.

[67] Gregoris Mentzas. Coupling object-oriented and workflow modelling in business and information reengineering. *Information Knowledge and Systems Management*, 1(1):63–87, 1999.

[68] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, NY, USA, 1988.

[69] John Mylopoulos. Private communication, 2006.

[70] John Mylopoulos. Object-oriented analysis. `http://www.cs.toronto.edu/~jm/2507S/Notes04/OOA.pdf`; accessed September 15, 2006.

[71] John Mylopoulos and Jaelson Castro. Tropos: A framework for requirements-driven software development. *In J. Brinkkemper and A. Solvberg, Editors, Information Systems Engineering: State of the Art and Research Themes. SpringerVerlag*, 2000. URL `citeseer.nj.nec.com/393259.html`.

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

[73] John Mylopoulos, Lawrence Chung, Stephen Liao, Huaiqing Wang, and Eric Yu. Exploring alternatives during requirements analysis. *IEEE Software*, 18(1):92–96, /2001. URL `citeseer.nj.nec.com/mylopoulos01exploring.html`.

[74] John Mylopoulos, Manuel Kolp, and Jaelson Castro. UML for agent-oriented software development: The tropos proposal. In *UML 2001 - The Unified Modeling Language.Modeling Languages, Concepts, and Tools: Fourth International Conference*, LNCS 2185, pages 422–441. Springer, 2001.

[75] John Mylopoulos, Manuel Kolp, and Paolo Giorgini. Agent-oriented software development. In *Methods and Applications of Artificial Intelligence: Second Hellenic Conference on AI, SETN*, LNCS 2308, pages 3–17. Springer, 2002.

[76] Bashar Nuseibeh and Steve M. Easterbrook. Requirements engineering: A roadmap. In Anthony Finkelstein, editor, *The Future of Software Engineering 2000*, June 2000.

[77] Kristen Nygaard and Ole-Johan Dahl. The development of the simula languages. In *HOPL-1: The first ACM SIGPLAN conference on History of programming languages*, pages 245–272, New York, NY, USA, 1978. ACM Press.

[78] Robert Pirsig. *Zen and the Art of Motorcycle Maintenance*. Bantam, reissue edition, 1984.

[79] Isabel Ramos, Daniel M. Berry, and Jo ao A. Carvalho. Requirements engineering for organizational transformation. *Journal of Information and Software Technology*, 47(5):479–495, May 2005.

[80] Awais Rashid, Bedir Tekinerdoğan, Ana Moreira, João Araújo, Jeff Gray, Jan Gerben Wijnstra, and Paul Clements. Early aspects: Aspect-oriented requirements engineering and architecture design. In *Workshop at AOSD-2002*, 2002. `http://trese.cs.utwente.nl/AOSD-EarlyAspectsWS/`; accessed January 31, 2006.

[81] Dale M. Rickman. A process for combining object oriented and structured analysis and design. In *Proceedings of NDIA 3rd Annual Systems Engineering & Supportability Conference*, October 2000.

[82] Walker Royce. *Software Project Management - A Unified Framework*. Addison-Wesley, Reading, MA, 1998.

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

[84] Stephen R. Schach. *Classical and Object-Oriented Software Engineering With Uml and Java*. McGraw-Hill, fourth edition, 1998.

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

[86] Sally Shlaer and Stephen J. Mellor. *Object-oriented systems analysis: modeling the world in data*. Yourdon Press, 1988. ISBN 0-13-629023-X.

[87] Guttorm Sindre. Private communication, 2006.

[88] Stephane 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.

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

[90] 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.

[91] Bhuvan Unhelkar. *Verification and Validation for Quality of UML 2.0 Models*. Wiley-Interscience, Hoboken, NJ, USA, 2005.

[92] Wil M.P. van der Aalst and Kees M. van Hee. Framework for business process redesign. In *Proceedings of the Fourth Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 95)*, pages 36–45, Berkeley Springs, 1995. IEEE Computer Society Press. URL `citeseer.nj.nec.com/vanderaalst95framework.html`.

[93] Axel van Lamsweerde. Requirements engineering in the year 00: A research perspective. In *Proceedings of 22nd International Conference on Software Engineering*, June 2000.

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

[95] 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.

[96] Michael von der Beeck. A comparison of statecharts variants. In *ProCoS: Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 128–148, London, UK, 1994. Springer-Verlag. ISBN 3-540-58468-4.

[97] 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.

[98] Roel Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Comput. Surv.*, 30(4):459–527, 1998. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/299917.299919.

[99] Edward Yourdon. *Modern Structured Analysis*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[100] Edward Yourdon and Larry Constantine. *Structured Design: Fundamentals*

of a Discipline of Computer Program and Systems Design. Prentice Hall, Englewood Cliffs, NJ, 1979.