A COMPARISON OF TWO FRAMEWORKS FOR MULTIPLE-VIEWED SOFTWARE REQUIREMENTS ACQUISITION

by

BINGYANG WEI

A DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in
The Department of Computer Science
to
The School of Graduate Studies
of
The University of Alabama in Huntsville

HUNTSVILLE, ALABAMA

2015

DISSERTATION APPROVAL FORM

Submitted by Bingyang Wei in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science and accepted on behalf of the Faculty of the School of Graduate Studies by the dissertation committee.

We, the undersigned members of the Graduate Faculty of The University of Alabama in Huntsville, certify that we have advised and/or supervised the candidate of the work described in this dissertation. We further certify that we have reviewed the dissertation manuscript and approve it in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.

Dr. Harry S. Delugach	(Date)	Committee Chair
Dr. Letha Etzkorn	(Date)	
Dr. Dan M. Rochowiak	(Date)	
Dr. Mary E. Weisskopf	(Date)	
Dr. Guo-Hui Zhang	(Date)	
Dr. Heggere S. Ranganath	(Date)	Department Chair
Dr. Sundar Christopher	(Date)	College Dean
Dr. David Berkowitz	(Date)	Graduate Dean

ABSTRACT

School of Graduate Studies
The University of Alabama in Huntsville

Degree	Doctor of Philosophy	College/Dept.	Science/Computer Science
Name of Candidate	Bingyang Wei		
Title	A Comparison of Two Frameworks for Multiple-viewed Software Requirements Acquisition		

The requirements process is a knowledge-intensive activity in which a large amount of requirements knowledge is acquired from different sources. Multiple-viewed requirements modeling facilitates that process by allowing modelers to observe the system from different viewpoints. Requirements knowledge is then organized and encoded in different requirements analysis models which collaboratively form an overall understanding of the system. One problem that always plagues modelers is the acquisition of requirements knowledge for building analysis models. An important but commonly neglected source of new requirements knowledge is the models that have already been built for the system under development. By translating each currently available model into a target model to be built, an incomplete target model is generated which then can be used as a modest spur to encourage modelers to provide more requirements knowledge. Two frameworks can be used to support such requirements acquisition: pair-wise and common representation frameworks. In practical applications, various factors need to be considered when requirements modelers choose between the two frameworks in order to acquire requirements by analysis model transformations. In this dissertation, we develop a conceptual graphs-based common representation framework for requirement knowledge acquisition, and then propose a set of criteria which provides a theoretical basis for comparing the two frameworks for their effectiveness of generating models and acquiring requirements in the context of multiple-viewed requirements modeling.

Abstract Approval:	Committee Chair	*Dr. Harry S. Delugach*
	Department Chair	*Dr. Heggere S. Ranganath*
	Graduate Dean	*Dr. David Berkowitz*

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to my advisor, Dr. Harry S. Delugach, for his encouragement and guidance. When I was uncertain about my future direction, Dr. Delugach encouraged me and was willing to become my advisor to guide me through this journal. I would like to thank the Computer Science Department for giving me the opportunity to study as a GTA to support my research. Special thanks to my dissertation committee members - Dr. Letha Etzkorn, Dr. Dan M. Rochowiak, Dr. Mary E. Weisskopf and Dr. Guo-Hui Zhang - for their valuable advice and effort on this work. I want to say thanks to Dr. Wei Li for his advice and encouragement when I first started to do research. I wish he will come back to the department soon.

Thanks to my dear friends, especially Dr. Yi Wang, Dr. Dong Wang, and Dr. Raouf Alomainy, for every moment we are together in the past five years.

I dedicate this work to my family for their love and support. I show my sincere thanks to my dear parents, Dr. Changjiang Wei and Mrs. Shaoxia Fu, for their understanding and support. My endless gratitude to my future-wife, Mengyuan Yu, I am so lucky to meet you and thank you for standing by me during this journey.

A NOTE ABOUT TYPEFACES AND COLORS

This work uses three different typefaces. Bulk of the document uses the same typeface that this sentence is written in. This is the regular typeface. When UML terms in a figure are introduced in the text, a bold typeface is used, like event `Pump Failure`. Lastly, an italic typeface, like *State: MethaneAlarmSwitchOn*, indicates that the item is from conceptual graphs.

This document uses five colors to differentiate the various types of requirements knowledge expressed by UML and conceptual graphs: UML class diagrams and the conceptual graphs that represent their semantics are in orange. UML state diagrams and the conceptual graphs that represent their semantics are in green. UML sequence diagrams and the conceptual graphs that represent their semantics are in cyan. Light gray is used in conceptual graphs to indicate inferred knowledge. Canonical graphs are in white.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS/ACRONYMS

ACRONYM	DEFINITION
CGs	Conceptual Graphs
PWF	Pair-wise Framework
CRF	Common Representation Framework
UnivSys.	University Information System
Cryptanlys.	Cryptanalysis
MineSys.	Mine Safety Control System

How do I know what I do not know?

—a desperate requirements modeler

CHAPTER 1

INTRODUCTION

Research is what I'm doing
when I don't know what I'm doing.

—Wernher von Braun

1.1 Overview

Software requirements describe the functions and constraints that a software system must have in order to achieve an objective within a specific application domain. Properly eliciting and modeling the requirements are critical to successful software development. Multiple-viewed requirements modeling is commonly used so that a system is observed from different viewpoints by different modelers and the requirements are expressed in different types of analysis models. Each analysis model is responsible for describing one aspect of the system and all of them together constitute the overall description of the system. Moreover, these models form an interesting requirements "ecosystem," where they evolve concurrently (Figure 1.1) by acquiring more requirements knowledge during the requirements process.

The building of different types of analysis models for a system is parallel and iterative by nature and might be conducted by different requirements modelers. However, it is difficult for a modeler to know whether an analysis model is complete

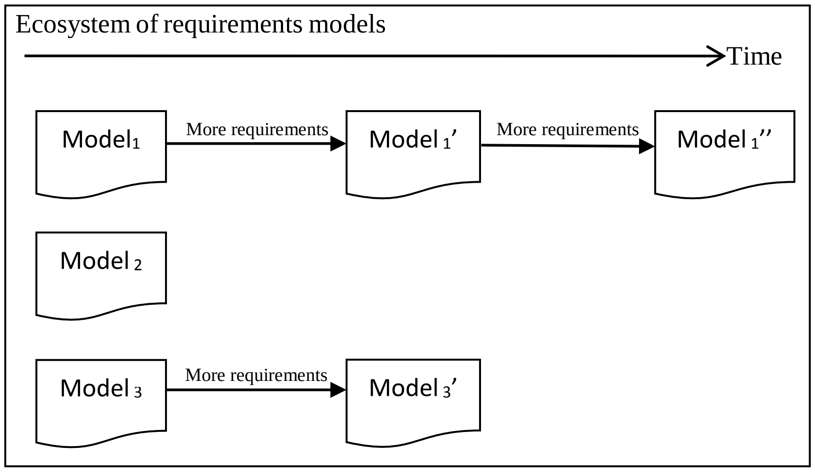
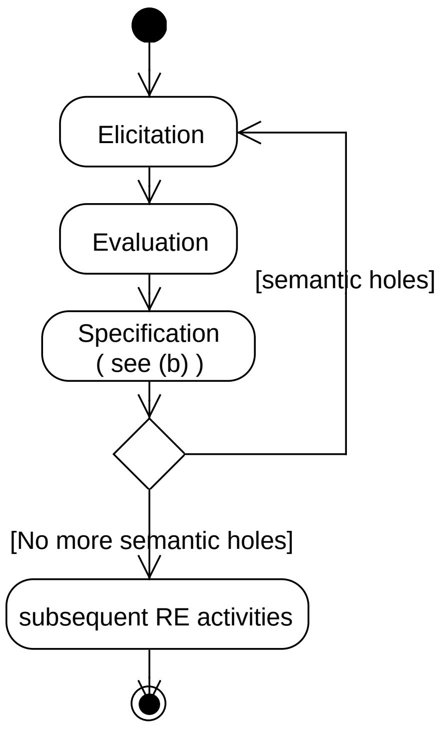


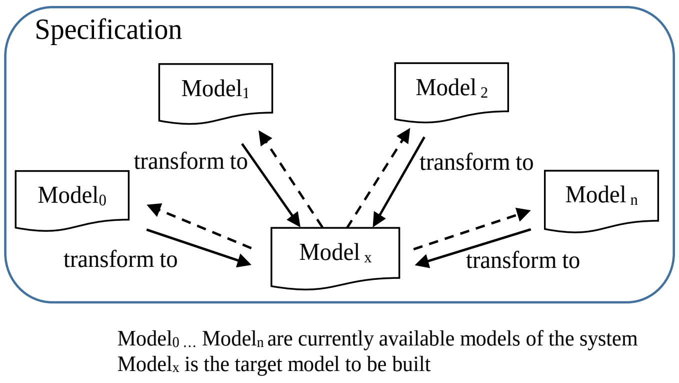
Figure 1.1: Ecosystem of multiple-viewed requirements modeling

or what requirements are missing from the current analysis model [1]. The same difficulty also exists when a modeler starts to build an analysis model at the very beginning. The modelers need to be made aware of the missing requirements of analysis models.

In order to make explicit the missing requirements in an analysis model, Delugach proposed the idea of conceptual feedback [2] which can provide prompts for the missing requirements to modelers (see Figure 1.2). This approach is based on the requirements knowledge overlap among analysis models of the same system. During conceptual feedback, requirements in the already-created analysis models ($Model_0$ to $Model_n$ in (b) of Figure 1.2) in the "ecosystem" are transformed to generate requirements needed for constructing a target analysis model $Model_x$. This process introduces new requirements to $Model_x$, which make it more complete, and, more importantly, causes it to generate some "semantic holes" which reveal some missing requirements that a modeler is not aware of before the generation process. A



(a)


Specification
Model1
Model 2
Model0
Model n
Model x
transform to
transform to
transform to
transform to
Model0 ... Modeln are currently available models of the system
Modelx is the target model to be built

(b)

Figure 1.2: Conceptual feedback in RE process

simple example of adopting the conceptual feedback approach to generate semantic holes in a UML class diagram is shown in Figure 1.3 and Figure 1.4. Through model transformations, the original UML class diagram shown in Figure 1.3 is augmented by requirements from the current state diagrams and sequence diagrams of a Mine Safety Control system (not shown here). The resulting augmented class diagram is shown in Figure 1.4. Note that the augmented class diagram contains semantic holes which denote the potentially missing requirements. In other words, they are requirements of the class diagram which cannot be automatically generated based on the existing requirements knowledge in other currently available models, therefore need to be explicitly provided by modelers. For example, class `PumpActuator` in Figure 1.4

has two semantic holes: an unspecified attribute and an unspecified operation, which graphically are shown as question marks in the attribute compartment and operation compartment, respectively.

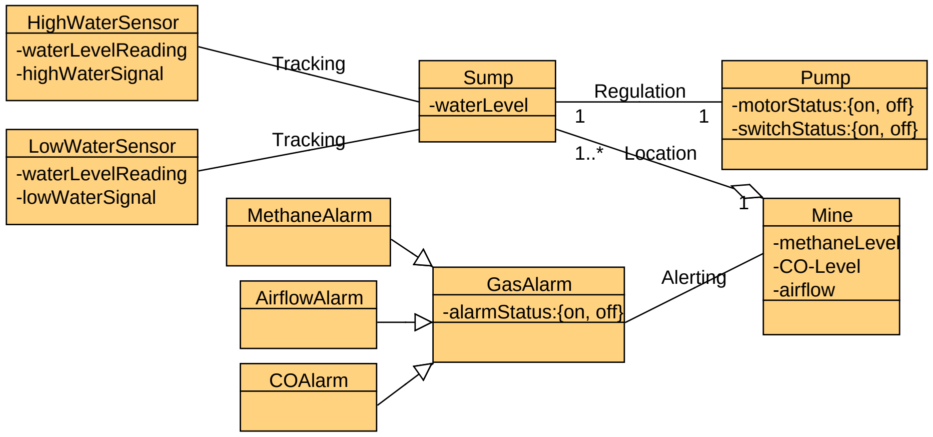


Figure 1.3: Original class diagram before model transformations

Because of the presence of semantic holes, the augmented analysis model is considered as incomplete so modelers are invited to complete it by providing the missing requirements. This enables additional new requirements to be elicited by filling in those holes (This process is shown as a backtracking from Specification to Elicitation in (a) of Figure 1.2). After eliciting requirements and filling in the semantic holes, the augmented model with semantic holes resolved would in turn affect other models (dotted arrows in (b) of Figure 1.2), causing further generation and completion processes in other analysis models of the "ecosystem." The process may repeat until no more new requirements knowledge can be acquired by transforming models, i.e., the "ecosystem" is internally complete and self-consistent.

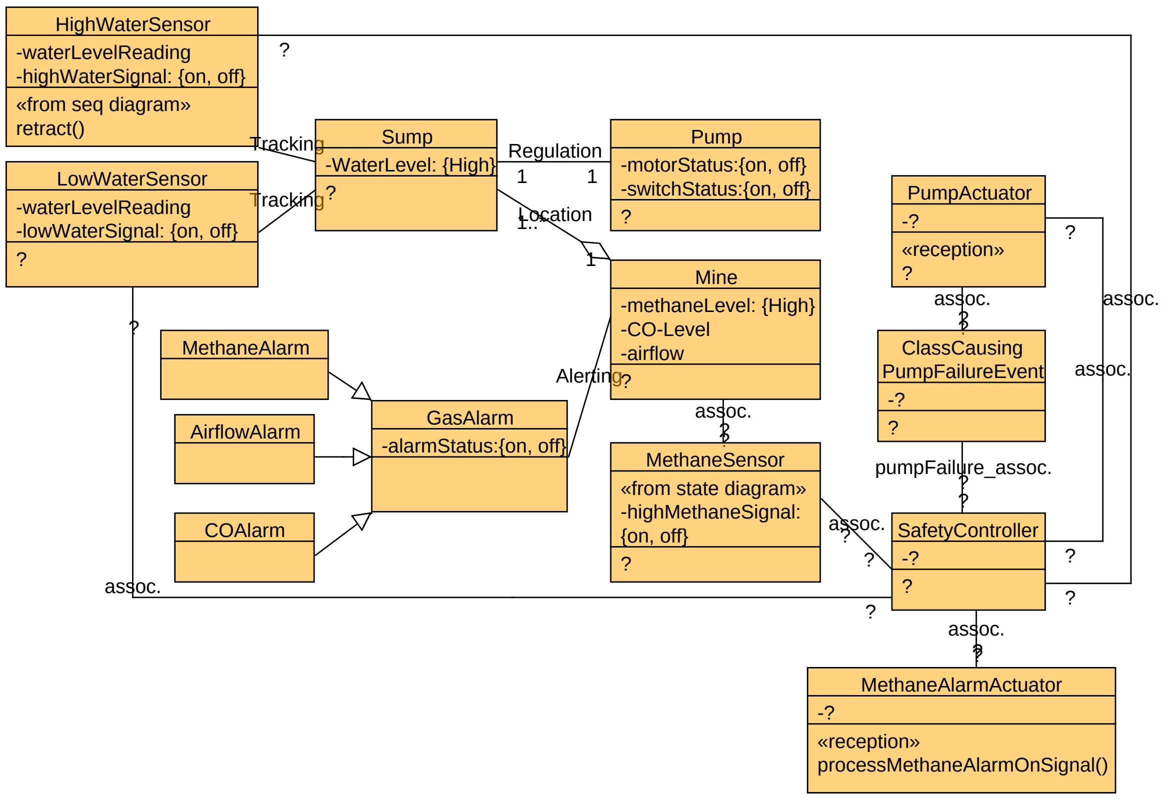


Figure 1.4: Augmented class diagram after model transformations

In summary, model transformations in a requirements "ecosystem" enable continuous knowledge flow between models, thus enhancing communication between different types of models and driving the requirements development process among individual models. This approach then will guide a team of requirements modelers to achieve a set of models which are consistent and complete.

1.2 Research Problem

In the context of multiple-viewed requirements modeling, there are two possible frameworks (Figure 1.5) that can support software requirements acquisition by model transformations. They are the pair-wise framework (PWF) and the common representation framework (CRF). The former framework supports requirements acquisition by direct transformations between models whereas the latter by replying on a common knowledge representation that describes the semantics of the analysis models in the "ecosystem."

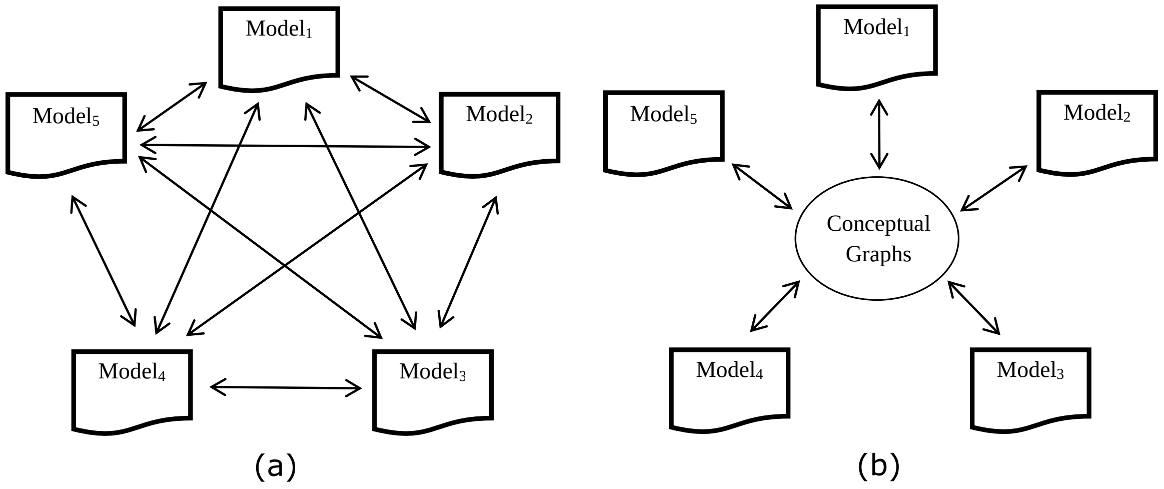


Figure 1.5: PWF and CRF

Both frameworks are widely used in the context of multiple-viewed requirements modeling, mainly for requirements consistency checking between models [3] [7] [8] [9] [10]. However they are not generally used for the purpose of requirements knowledge acquisition. For PWF, Selonen et al. [4] described the semantics of pair-wise model transformations between UML diagrams and indicated that such transformations could be used to produce an initial incomplete model for knowledge acquisition purposes; for CRF, Delugach [2] proposed using conceptual graphs (CGs) [11] in CRF in order to transform and acquire requirements for analysis models. However the semantics of the transforming models to and from CGs were only partially defined and no explicit guidelines were described. A more detailed and semantically solid description of model transformations in the CRF used for requirements acquisition is needed to greatly enhance the quality of requirements acquisition process.

Another important research problem is that the two frameworks' effectiveness in transforming model and acquiring requirements knowledge is largely unknown. There is no theoretical basis for evaluating which framework is more effective in transforming models and acquiring requirements in the context of multiple-viewed requirements modeling. A detailed analysis of the capability of eliciting requirements from requirements modelers, the capability of preserving semantics and the extensibility of each framework, and an objective comparison is needed so that researchers and modelers can rely on this proposed criteria and comparison to choose between frameworks for addressing requirements acquisition problem.

In summary, this dissertation proposes to develop a formal semantic description of requirements acquisition in CRF and to comes up with a comparison methodology which can be used to evaluate PWF and CRF for their effectiveness in acquiring requirements knowledge in a multiple-viewed requirements modeling context.

1.3 Contributions

This work contributes to software engineering community and knowledge representation community in two ways:

1. Develop the semantic description of the requirements knowledge acquisition in the CGs-based CRF to demonstrate that knowledge-based view transformations

can indeed provide effective assistance to the critical requirements process by acquiring more requirements knowledge.

2. Provide a set of useful criteria for comparing frameworks that can address the requirements acquisition problem in multiple-viewed modeling so that modelers can reference the criteria when choosing frameworks.

1.4 Outline of the Dissertation

The remainder of the dissertation is organized as follows: Chapter 2 surveys previous representative work which adopts one of the two frameworks in multiple-viewed modeling; CGs are briefly introduced as well. In Chapter 3, we present an overview of the two frameworks (PWF and CRF) that support requirements knowledge acquisition and propose the comparison strategy. Detailed model transformations in the two frameworks are elaborated in Chapter 4 and Chapter 5, respectively. In Chapter 6, the two frameworks are evaluated according to a set of criteria; the results of this comparison are presented. Chapter 7 discusses the strengths and limitations of this work. Chapter 8 concludes the dissertation.

This dissertation evaluates the requirements acquisition in the two frameworks and compares them based on three complete complete case studies in Chapter 6. Because of their size, the results are presented in the appendices.

CHAPTER 2

BACKGROUND

This chapter presents the necessary background for the work. Section 2.1 surveys previous representative work that adopts one of the two frameworks to transform analysis models during multiple-viewed requirements modeling, with the aim of identifying their transformation purposes, strategies, and relevance to our work; we also provide the inspiration and foundation for the common representation framework that this work is going to develop. Conceptual graphs are then introduced in Section 2.2.

2.1 Literature Review

Model transformations in the context of multiple-viewed requirements modeling have been studied by a number of other researchers over the years. This section reviews literature related to our work; literature using PWF and CRF are described in Section 2.1.1 and Section 2.1.2, respectively. Since one of the contributions of this work is to develop a semantic description of the CRF for requirements acquisition, the difference between related work and our work is also discussed in Section 2.1.2;

important research on requirements knowledge that inspires our work is presented in Section 2.1.3.

2.1.1 Previous Work Using PWF

Model transformations using PWF are common and straightforward where direct transformation rules are defined for each pair of models [3]. In multiple-viewed requirements modeling, n types of models would therefore require $n(n-1)$ transformation rules.

Among the work using PWF, Nuseibeh et al. [3] proposed ViewPoint, which bundles representation scheme, view development process, and partial specification together. Inter-ViewPoint rules were defined when building ViewPoints. By specifying the rule application mode (check mode or transfer mode), the rules were applied to check consistency between a pair of ViewPoints, or to transform information in one ViewPoint's partial specification to another. The transfer mode in application of inter-ViewPoint rules was used as a way to resolve inconsistencies, not a way to expose incompleteness for the purpose of acquiring requirements knowledge.

In [7], Egyed compared heterogeneous views for detecting inconsistency by direct view-to-view transformation since transformation makes a view more understandable in the context of other views so that heterogeneous views become more easily comparable. Transformation rules were defined to make the transformation meaningful and accurate. The purpose of his model transformation in PWF was for model comparison so that inconsistency can be detected.

Selonen and his colleagues [4] investigated the relationships among different types of UML diagrams and discussed transformation operations that are based on those relationships. Their work indicated that one of the uses of model transformations in PWF is model synthesis. That is, producing a requirements analysis model on the basis of an existing requirements analysis model of another type. In our work, model synthesis is only one of our purposes, we focus more on exposing incompleteness in the generated analysis models and using such incompleteness as a way to acquire more requirements from human modelers. Still, the semantic descriptions of transformation operations between UML diagrams in their work are of great value to us; the PWF that we are going to build in Chapter 4 uses some of their transformation rules.

There is more previous work dealing with transformation between models, most of which is about transformations among UML diagrams. For example, the research in [12] and [13] transformed use case diagrams to other UML diagrams; Koskimies et al. [14] transformed sequence diagrams to state diagrams; object-to-class transformations were studied in [15]. Table 2.1 summarizes the related work using PWF.

Table 2.1: Previous work using PWF

Previous work	Analysis models	Purposes
Nuseibeh et al. [3]	ViewPoints	Inconsistency detection and model generation
Egyed [7]	UML diagrams	Inconsistency detection and model comparison
Selonen et al. [4]	UML diagrams	Inconsistency detection and model synthesis
Liu et al. [12]	UML diagrams	Use cases to other UML diagrams conversion
Koskimies et al. [14]	Sequence and state diagrams	Sequence diagrams to state diagrams conversion
Engels et al. [15]	Object and class diagrams	Object diagrams to class diagrams conversion

2.1.2 Previous Work Using CRF

An alternative to pair-wise transformation of models is the use of a common representation as an intermediate representation. In CRF, different types of models are converted to a common representation so that different kinds of analysis (e.g. consistency checking, integration, completeness checking etc.) can be conducted with the same representation. This framework also reduces the number of transformation rules to $2n$ for models in n different representational styles. In this subsection, we list several related works adopting CRF during the requirements analysis process.

CGs have been used as a common representation to express requirements analysis models in [2] [6] [8] [16]. Delugach [2] proposed a CGs-based CRF for validating and acquiring requirements knowledge in multiple-viewed requirements modeling. In his work, Delugach converted OMT object diagrams and data flow diagrams to CGs where their semantic relationships were described. The combined CGs were then projected back to each originating model, allowing a requirements modeler to acquire new information in his original notation. The requirements modeler could then examine, validate, and evaluate the new information from other views in his own view notation. In Cyre's work [6], block diagrams, flowcharts, timing diagrams and natural language requirements specifications were converted to CGs for integration, consistency, and completeness analysis, and for further automatic synthesis. Our work is greatly inspired by Delugach and Crye's work. However, in [2], the semantics of the requirements models were only partially described and no systematic process of converting and generating models was provided; and in [6], after converting different types of models to CGs, no further discussion was provided on requirements acquisition through transforming CGs back to models. The CRF developed in this dissertation builds on Delugach and Cyre's research; it is an extensible framework based on the CGs Support which facilitates transforming different types of analysis models to and from CGs for the purpose of requirements knowledge acquisition.

The work of Sunetnanta [8] and Thanitsukkarn [16] focused on model consistency checking using CGs. Models were converted to CGs and checked against consistency rules which were also expressed in CGs. However, conversion strategy used in their work was different from Delugach and Crye's. In [8] and [16], CGs were used as a meta-representational language, so the resulting combined CGs just captured the structural information of analysis models. We call this conversion strategy "Structural conversion" and find out that structural conversion cannot be effectively used for knowledge acquisition tasks in our work.

Whilst the use of CGs as a common representation worked well in the above literature, there is nothing inherent to CGs that makes them the only way for realizing a CRF. Shan and Zhu [17] tried to formally specify the semantics of UML diagrams in first-order logic for the purpose of requirements consistency checking. Lucas et al. [18] managed the requirements consistency problems using two formalisms: a transformation language QVT Relation [19], which was used to define consistency relationships between meta-models of requirements analysis models, and a rewriting logic Maude [20], which gave support for checking consistency. Both models and QVT Relations were finally transformed into Maude. Van Der Straeten et al. [9] translated UML diagrams and consistency constraints between diagrams into description logic [21]. This approach could also fix some errors by inserting missing requirements. However, as with the previous work [8] [16] which use CGs as central formalism, they focused on transforming the structural information of requirements analysis models based on meta-models of UML diagram. The use of common representation in those works does not effectively capture the semantics behind each analysis model and, as a result, such use of common representation does not fit in the requirements acquisition task in our work.

Besides the purposes of requirements acquisition and requirements consistency checking using CRF, other research used CRF to generate models: Jaramillo et al. [22] presented pre-conceptual schema, a CG-like knowledge representation with certain additional symbols representing dynamic properties, and used it in the requirements analysis task. The framework they provided could automatically construct pre-conceptual schemas from requirements written in a controlled natural language and then derive three types of UML diagrams (class, state and communication diagrams). However, our work is not trying to acquire knowledge directly from textual requirements but from diagrammatic models like UML diagrams, and emphasizes more the underlying relations between models in the same "ecosystem" so that requirements knowledge can flow across all analysis models. Hasegawa and his colleagues [23] extracted requirements knowledge from Japanese textual requirements and represented it in CGs. The resulting CGs could be used to derive models. The authors predefined a meta-model of CGs and the textual requirements then would be interpreted in terms of the predefined meta-model. This meta-model was general enough to be used everywhere. However, the resulting CGs model could only be used to construct class diagrams, so this approach did not capture requirements knowledge for other types of diagrams in UML. Our approach covers more analysis models and thus is more comprehensive and useful. Both [22] and [23] demonstrate the capability of CGs in representing requirements knowledge and their semantic analysis of requirements greatly benefit our work.

Previous work using CRF during the requirements process is summarized in Table 2.2. The key differences between previous work and our work are the purposes of the conversions and the way that models are converted to a common representation (see Purpose and Conversion strategy in Table 2.2).

Table 2.2: Previous work using CRF

Our work and previous work	**Knowledge representation used**	**Purpose**	**Conversion strategy**
Our work	CGs	Requirements acquisition	Semantic
Delugach [2]	CGs	Requirements acquisition	Semantic
Cyre [6]	CGs	Inconsistency and completeness detection	Semantic
Jaramillo et al. [22]	Modified CGs	UML diagram generation	Semantic
Sunetnanta and Finkelstein [8]	CGs	Inconsistency detection	Structural
Lucas et al. [18]	Maude	Inconsistency detection	Structural
Van Der Straeten et al. [9]	Description logic	Inconsistency detection	Structural
Shan and Zhu [17]	First-order logic	Inconsistency detection	Structural

In Table 2.2, the work in [6] [8] [9] [17] converts models to a knowledge representation for the purpose of consistency checking among models. Jaramillo et al. convert textual requirements in CGs in order to generate UML diagrams. By contrast, our purpose in converting existing models into CGs is to generate initial UML diagrams with semantic holes which can facilitate requirements knowledge acquisition. While different, we see our work as a useful complement to these approaches.

Our approach is also different from previous work with respect to the conversion strategy. Most previous work performs a purely structural (essentially syntactic) conversion. This means that, in their resulting knowledge reservoir, there is not much additional requirements knowledge, just requirements knowledge in another form. As a result, even though different kinds of models are all converted into the same knowledge representation, the degree of integration of requirements knowledge in the resulting knowledge reservoir is low. These structure-oriented conversions work well for the purpose of consistency checking or matching for reuse but are not suitable for our knowledge acquisition purpose. Therefore, we adopt a semantic-oriented conversion approach as in [2] [6] [22]. However, the conversion process in our work is founded on the fundamental semantics of the typical object-oriented models as determined by [25] [26] (see next subsection). Our conversions depend on the set of primitive concepts and relations that underlies the requirements, which results in a more fine-grained and cohesive central requirements knowledge reservoir.

2.1.3 Requirements Knowledge

Research on studying the knowledge of software requirements is briefly surveyed in this subsection. In [25], Dardenne and Van Lamsweerde defined a conceptual meta-model for capturing requirements. The meta-model is based on a variety of abstractions usually found in software requirements documents. Such abstractions include classical concepts that already appear in many existing specification languages and new concepts like goals, constraints which have been introduced in KAOS (Knowledge Acquisition in automated specification). This meta-model is aimed at being sufficiently rich to cover all of the requirements that need to be acquired.

Davis and his colleagues [26] proposed a similar set and tried to define a set of primitive requirements elements, bonds, and composing rules that underlies all requirements views. In light of the above idea, requirements models of a system can be expressed in terms of the elements and bonds from the set following the composing rules. In this way, different kinds of views are combined and related together and changes made in one view are propagated to all other views which are defined based on the same set of primitives.

The work of Diskin et al. [10] defined the overlaps among meta-models of different types of requirements analysis models (class diagram, state diagram and sequence diagram) in an overlapping meta-model. Individual diagrams can be projected into the space of the overlapping meta-model and then merged together. They argued that this predefined central overlapping meta-model is "unavoidably minimal" since it only defines the necessary common view of each type of requirements analysis model.

The CRF in our work is based on the inspiration from the above work. The CRF we are developing in this work maintains a CGs Support where the primitive concepts and relations underlying the object-oriented requirements analysis models are defined (see details in Section 3.3). In this way, different types of models can be translated to and from CGs.

2.1.4 Summary

In this literature review section, we first reviewed related work using PWF and CRF in the context of multiple-viewed requirements modeling, and then discussed research on requirements knowledge that inspires our work.

For PWF, the research in [3] [4] [7] collaboratively carried out a well-understood semantic description of the framework. Selonen et al. [4] also mentioned that pair-wise view transformations could be used to produce an initial incomplete view so that the modeler could start to fill in the missing requirements.

For CRF, research is divided into two camps based on different conversion strategies (see Conversion strategy column in Table 2.2). The majority of previous work using CRF is for the purpose of checking consistency among requirements analysis models and generating analysis models. CRF is not generally used for the purpose of generating incomplete models in order to acquire more requirements knowledge. Our work will focus on this purpose and will develop a complete description of transforming UML diagrams in CRF using CGs as the common representation.

2.2 An Overview of CGs

This section provides a brief introduction to CGs, the central formalism we adopt in CRF in this work.

2.2.1 Philosophical Foundation

CGs are powerful knowledge representation formalisms which were developed by John F. Sowa from the existential graphs of Charles Sanders Peirce [27] and the semantic networks [28] of artificial intelligence. Based on existential conjunctive first-order logic, CGs express semantics in a form that is unambiguous and precise. Moreover, CGs' simple graphical notations with direct mapping to and from natural languages make CGs expressive and easy to comprehend. As a result, CGs are commonly used as a knowledge representation and can serve as an intermediate interpretation language between natural language and computer-oriented formalisms. Another important feature of CGs systems is that they have open-world semantics, which allows the specification of incomplete knowledge. As a result, CGs are well suited to express the requirements of a software application. The models of a software system can be regarded as a collection of statements that evaluate to truth, i.e., assertional knowledge. For the CRF we are developing in this work, we adopt this simple but competent representation to provide the basis for preserving semantics and automatic transformations of UML diagrams.

2.2.2 Basics

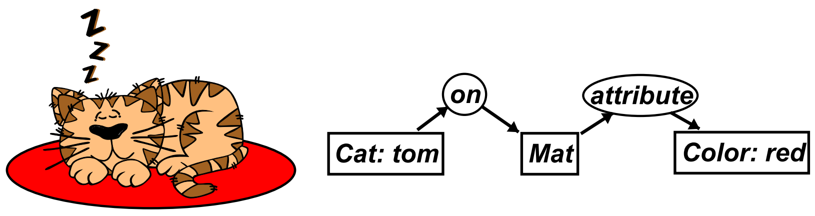


Figure 2.1: A picture and its description in CGs

Two basic elements of CGs are concepts and relations. Concepts are represented by rectangles and relations are represented by ovals (Figure 2.1). A concept in CGs means the existence of an instance of a thing, which may be an entity, an idea, a proposition, or any other thing. A concept node is labeled by a concept type e.g. *Cat*, *Mat*, *Color* and, optionally, by a referent e.g. *tom*. In Figure 2.1, there exist three concepts: a cat named tom, a mat, and a color. The second basic element is a relation which represents a relationship between existing concepts. The arcs that link the relations to the concepts are denoted by arrows: the direction of the arrows indicates the sense of the relation. In the above CGs, the relation *on* relates tom to a mat and the relation *attribute* relates that mat to a red color. The CGs in Figure 2.1 can be translated to a statement of the following form in typed predicate calculus:

$(\exists x : Cat, \exists y : Mat, \exists z : Color) Name(x, \text{“}tom\text{”}) \wedge Name(z, \text{“}red\text{”}) \wedge On(x, y) \wedge Attr(y, z)$

A concept by default asserts the existence of one thing. By adding a universal quantifier, a concept can represent all of the instances of a certain type (Figure 2.2).

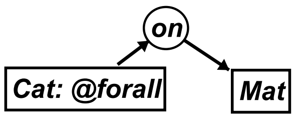


Figure 2.2: “Every cat is on a mat” in CGs

The types of concepts in a CGs system are organized in a type hierarchy ((a) Figure 2.3). The idea of a type hierarchy is the same as the class hierarchy in object-oriented modeling: sub-types inherit characteristics from their super-types, forming an “is-a-kind-of” relationship among them. Similar to the concept type hierarchy is a hierarchy of types of relations ((b) in Figure 2.3). Please note that a concept type is represented as a dotted rectangle with a bar at the bottom, while a relation type is represented as a dotted oval with a bar at the bottom.

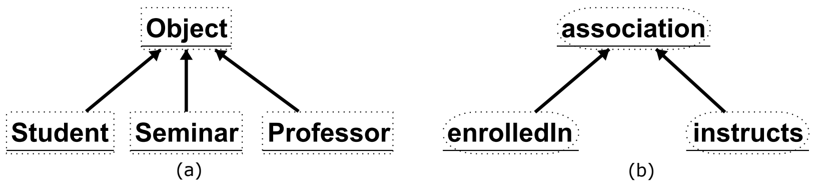


Figure 2.3: Concept type hierarchy and relation type hierarchy

Since CGs are a form of graphical logic, another important feature of CGs is their capability for visualized logic inference, which gives us the ability to infer knowledge that is implicitly present in the knowledge reservoir. Inference is supported in part by rules. One kind of inference rule has the form "If proposition1 then proposition2." The conceptual graph in (a) of Figure 2.4 means "If prop1 exists, then prop2 also exists."

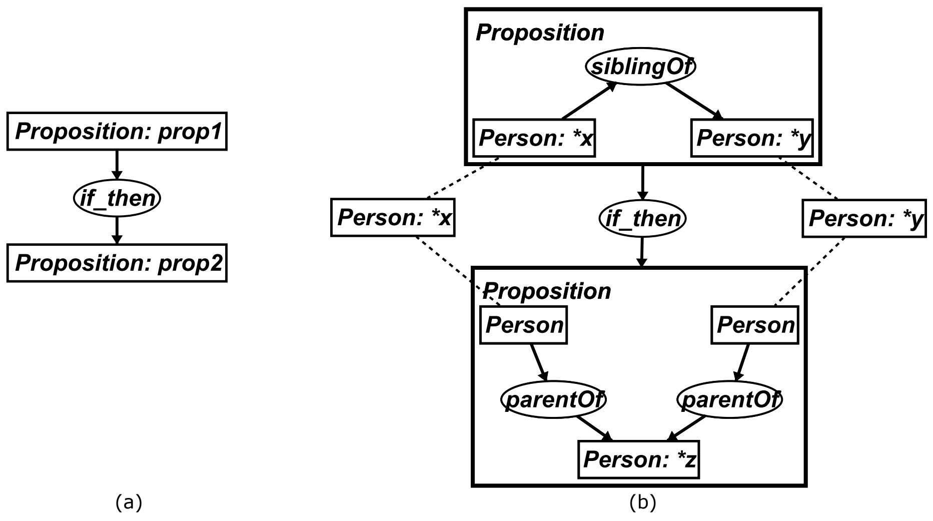


Figure 2.4: Inference rule in CGs

For example, the inference rule "If two persons are siblings, then there exists another person who is their parent." can be represented by CGs in (b) of Figure 2.4. It is a graphical transformation that essentially accomplishes a logical inference.

A context is a concept with the nested CGs as its referent. In Figure 2.5, the concept of type *CallEvent* is a context that describes a call event in a software system.

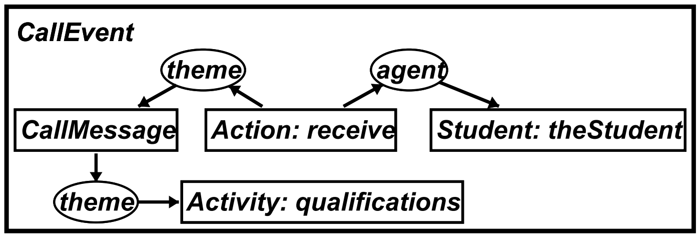


Figure 2.5: Context in CGs

The dashed line in (b) of Figure 2.4, called a co-reference link, denotes that the concept *Person* in the consequent proposition refers to the same individual as the concept *Person: *x* in the antecedent proposition box.

CHAPTER 3

OVERVIEW OF OUR APPROACH

In this chapter, we present an overview of our research approach so that readers can quickly get the overall idea of this work before diving into the details in the following chapters. Section 3.1 and Section 3.2 briefly describe the PWF and CRF that are developed in this work; Section 3.3 presents the comparison strategy.

In this work, three types of UML diagrams are considered in both frameworks (Figure 3.1). They are class diagrams, state diagrams, and sequence diagrams which are among the most commonly used diagrams in UML for specifying an object-oriented system. The reason for choosing them is that the structure, state, and interaction views of a system provide a sufficiently broad range of the semantics of object-oriented models to show the generality of our approach.

3.1 The Pair-wise Framework

As already mentioned in Chapter 2, relations between each pair of UML diagrams have been studied by many researchers and they collaboratively carried out a well-understood semantic description of model transformations in PWF. Particularly, Selonen et al. [4] mentioned that pair-wise model transformations could be used to

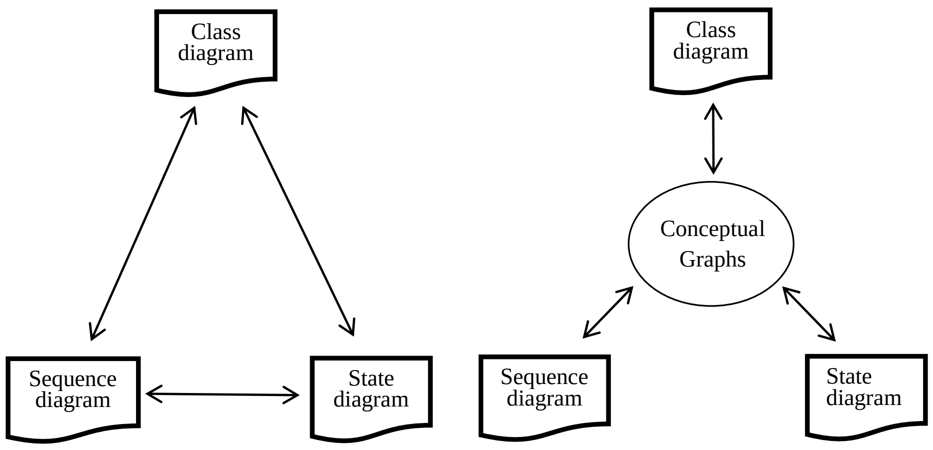


Figure 3.1: Three UML diagrams in two frameworks

produce an initial incomplete model so that the modeler could start to fill in the missing requirements. So the PWF in this work is developed according to transformation rules from previous works. These rules between each pair of the three UML diagrams are elaborated in Chapter 4.

3.2 The CGs-based Common Representation Framework

In this section, we present an overview of our CGs-based common representation framework. The framework consists of a CGs Reservoir where requirements knowledge of UML diagrams is stored in CGs form and a CGs Support which guides the process of converting and generating UML diagrams to and from CGs (Figure 3.2).

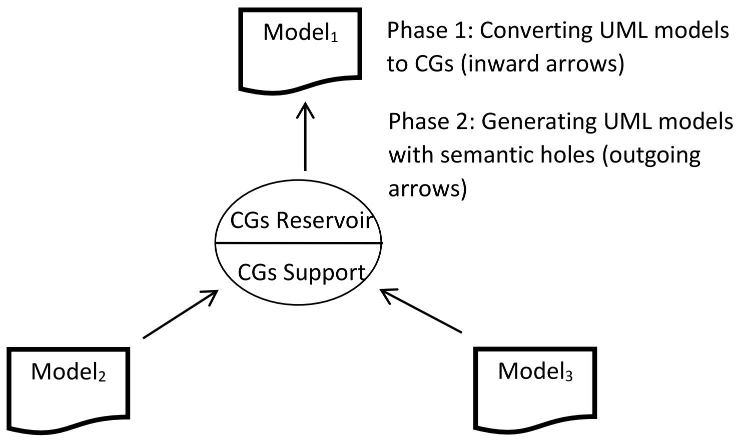

Model₁
Phase 1: Converting UML models to CGs (inward arrows)
Phase 2: Generating UML models with semantic holes (outgoing arrows)
CGs Reservoir
CGs Support
Model₂
Model₃

Figure 3.2: CGs-based CRF at work

The process of generating UML diagrams with semantic holes consists of two major phases: Phase 1: converting already developed UML diagrams to CGs in order to populate the CGs Reservoir (inward arrows in Figure 3.2); phase 2: generating a specific type of UML diagram from the CGs Reservoir (the outgoing arrows). For example, in Figure 3.2, $Model_2$ and $Model_3$ that have already been developed are converted to CGs. Based on the requirements knowledge in the CGs Reservoir and inference rules in the CGs Support, $Model_1$ is generated with semantic holes for requirements acquisition purposes. Detailed translations of UML diagrams to and from CGs are elaborated in Section 5.1 and Section 5.2 in Chapter 5, respectively. A requirements knowledge acquisition process then starts in which modelers provide necessary requirements to fill in the holes in the generated $Model_3$, thereby resolving the missing requirements.

The rest of this section elaborates upon a key component of the framework, the CGs Support.

3.2.1 The CGs Support

The CGs Support is a key component in our CRF. It defines, in CGs form, semantic elements called primitives as building blocks of the three UML diagrams, canonical graphs which are used to express the three UML diagrams in CGs, and inference rules for generating UML diagrams from the CGs Reservoir.

3.2.1.1 The Primitives

As discussed in Chapter 2, software engineering researchers have tried to identify the minimal set of fundamental elements that underlies the requirements of an object-oriented system [25] [26]. In the light of their work, the CGs Support of our CRF defines a set of elemental concepts and relations underlying the three UML diagrams so that any requirement captured by the three UML diagrams can be expressed in terms of the primitives. The primitives are to UML modeling languages as assembly language statements are to high-level programming languages. The types of primitive concepts and relations are organized in a CGs concept type hierarchy (Figure 3.3) and a CGs relation type hierarchy (Figure 3.4), respectively.

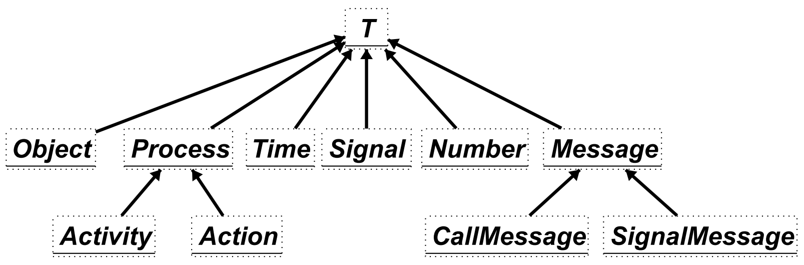


Figure 3.3: Primitive concept type hierarchy

The types of the primitive concepts in the CGs Support are *Object*, *Activity*, *Action*, *Message*, *Time*, *Signal* and *Number*. In CGs, all concept types have a common super type *T*. The *Object* type is the general description of something in a software system that has state and behavior, class of every object in a software system is either a direct or indirect descendant of the *Object* type. For example, classes `Student` and `Seminar` in the University Information System are subtypes of *Object* type when expressed in CGs. The *Activity* type is used to describe all behaviors performed by objects in a software system. For example, enroll in a seminar and get a student's schedule are instances of the type *Activity* when expressed in CGs. The *Action* type describes the smallest computation unit, such as issuing and receiving messages. The *Message* type describes information exchanged between objects in a software system (note that it has two subtypes, *CallMessage* and *SignalMessage*). The *Time* type describes times in a software system. The *Signal* and *Number* type describe all the signals and numbers, respectively.

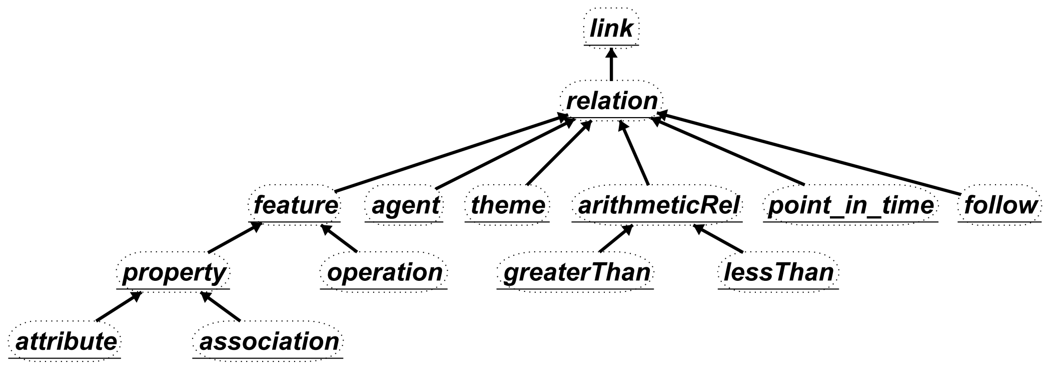


Figure 3.4: Primitive relation type hierarchy

Primitive relations relate primitive concepts to represent meaningful relationships among them. The types of primitive relations in the CGs Support are organized in a CGs relation type hierarchy (Figure 3.4). Figure 3.5 shows some meaningful CGs snippets composed of primitive concepts and relations.

An *attribute* relation relates an *Object* type concept to a *T* type concept (see (a) in Figure 3.5); the referent of the *T* type concept denotes an attribute value of the object represented by the *Object* type concept. An example of this is *Color: red* or *Size: large*. An *association* relation relates two *Object* type concepts ((b) in Figure 3.5), and it represents the semantic relationship that can occur between two objects in a software system. For example, a `Student` object is associated with a `Seminar` object by `enrolledIn` association. An *operation* relation relates an *Object* concept to an *Activity* type concept, and it depicts the relationship between an object and its operation. The CGs (c) in Figure 3.5 means an object has the ability of performing a certain operation. An *agent* relation relates an *Object* type concept and an *Activity* type concept ((d) in Figure 3.5), and it also describes the relationship between an object and an operation, but it means that the operation is actually performed by an object during the execution of the system. A *theme* relation is used to associate a part to the main part ((e) in Figure 3.5), such as the arguments of an operation and the content of a message. The *theme* relation is different from the *attribute* relation in that a *theme* relation never relates *Object* type concepts. Relation between *Number* type concepts is represented by *arithmeticRel* relation (see (f) in Figure 3.5). A *point_in_time* relation relates a *Proposition* context to a *Time* type concept. A *follow* relation connecting two situations expresses the meaning of one starts after the previous situation is done (see (g) in Figure 3.5). By introducing the *point_in_time* and *follow* relations in our primitives, we can use a monotonic logic like CGs to represent dynamic semantics in state and sequence diagrams.

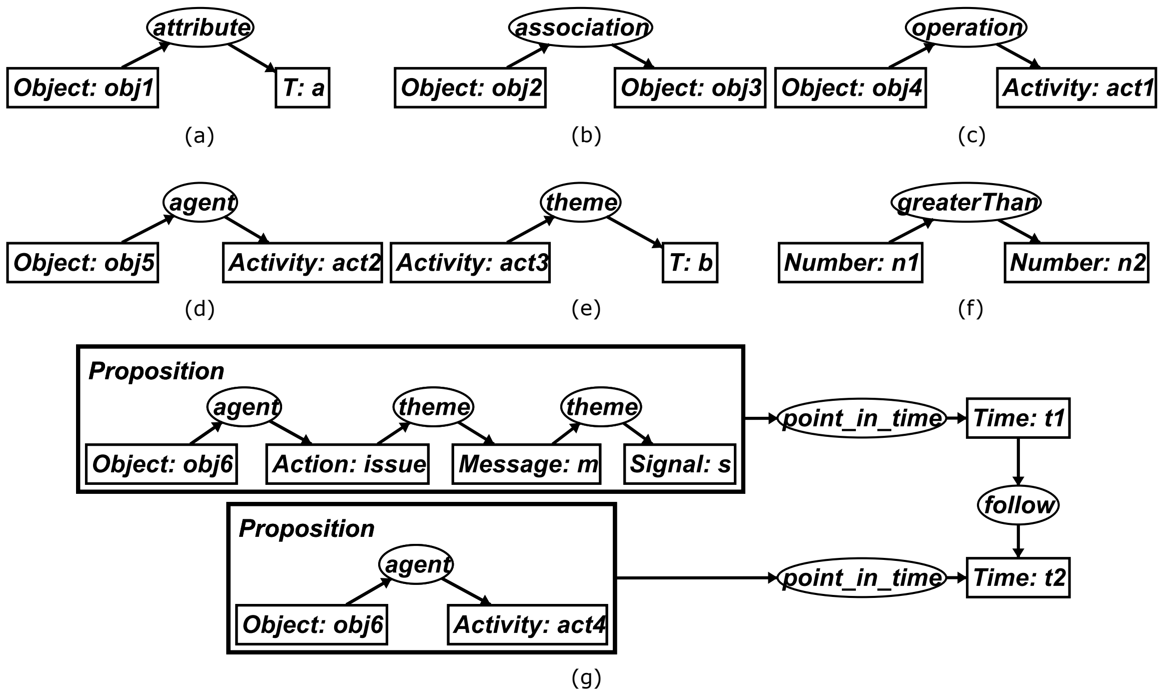


Figure 3.5: Meaningful CGs made up of primitive concepts and relations

Tthe primitive concepts and relations cannot be further decomposed (they are the smallest semantic unit in CRF).

3.2.1.2 The Canonical Graphs

The conversion of UML diagrams to CGs in phase 1 is based on special CGs called canonical graphs. Each type of UML diagram has a corresponding set of canonical graphs that describes its semantics in CGs using primitives. Canonical graphs are CGs used as templates to represent meaningful relationships among concepts in a particular type of UML diagram. The canonical graphs are not models themselves. For example, several different UML class diagrams can be converted to different CGs using the same set of canonical graphs. UML diagrams are converted into CGs by instantiating corresponding sets of canonical graphs so that all of the semantics captured by the UML diagram are preserved in the central CGs Reservoir. For example, a canonical graph that represents semantics of a state transition is shown in Figure 3.6. In Chapter 5, canonical graphs of class, state, and sequence diagrams are defined and used to convert the three different types of UML diagrams into CGs while still preserving the semantics of the originating UML diagrams.

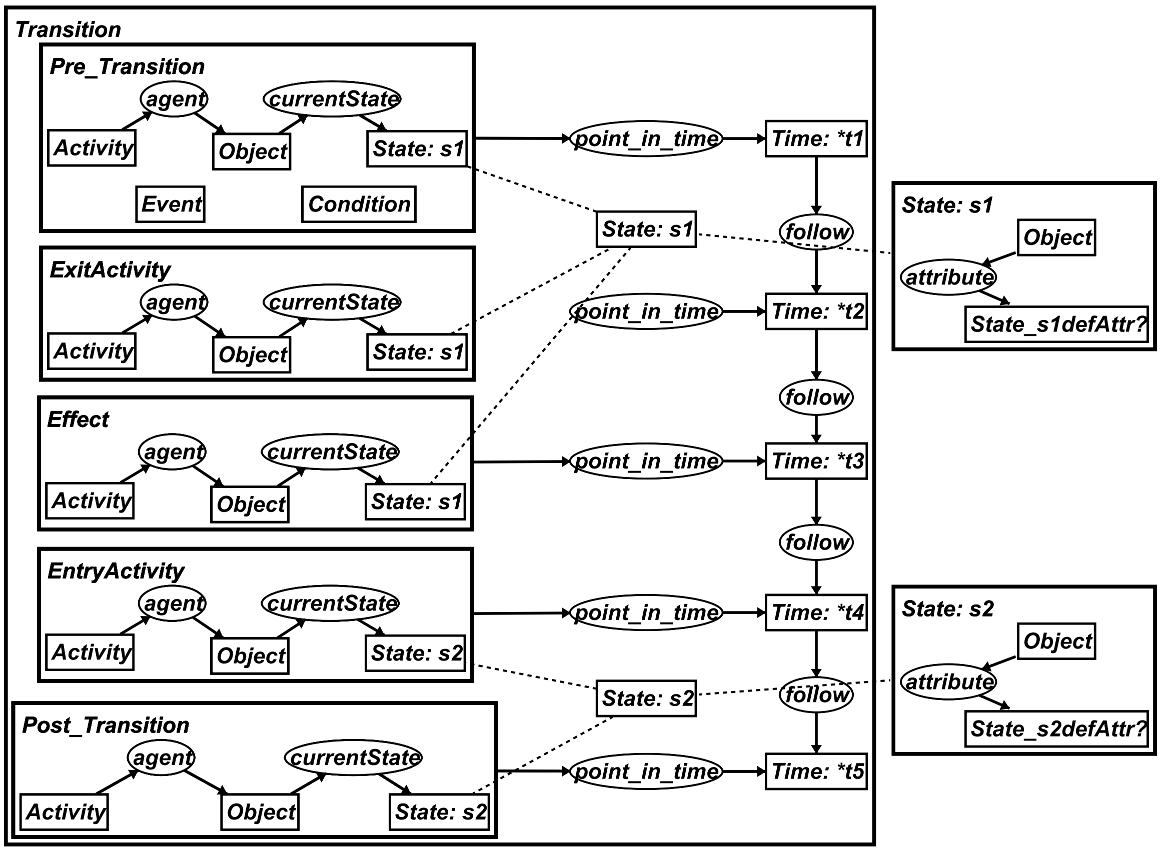


Figure 3.6: Canonical graph of a state transition

3.2.1.3 The Inference Rules

The CGs Support contains rules which are used to infer requirements knowledge for generating UML diagrams. The generation process is based on a forward-chaining inference method. As in any logical inference, the presence of a rule's antecedent in the CGs Reservoir implies its consequent which represents the desired requirements knowledge used to build a target UML diagram. During this process, for each inference rule of the target UML diagram, the CGs Reservoir is scanned to look for CGs snippets that match the antecedent of the rule. If a match is found, the consequent of the rule is asserted, thereby resulting in the derivation of requirements knowledge needed for building the diagram.

To illustrate this, an inference rule of UML state diagrams is shown in Figure 3.7. When applying this rule, the CGs snippet we are looking for in the CGs Reservoir is "an *Object* type concept receives a *Message* type concept," which, if found, would imply both the existence of an *Event* type concept (*Event* concept in Figure 3.7) and the fact that the aforementioned *Object* type concept is related to a certain *State* concept (*State* concept in Figure 3.7) through *currentState* relation. Note that the inferred CGs are colored in light gray in Figure 3.7. Since both *State* and *Event* concepts are not primitives, they must be defined in terms of primitive concepts and relations (see *State* and *Event* context in Figure 3.7). Such inferred requirements knowledge is useful for building a UML state diagram. This process continues until all inference rules of state diagrams have been applied and no more facts can be inferred. Then a state diagram can be built based on the requirements knowledge thus obtained.

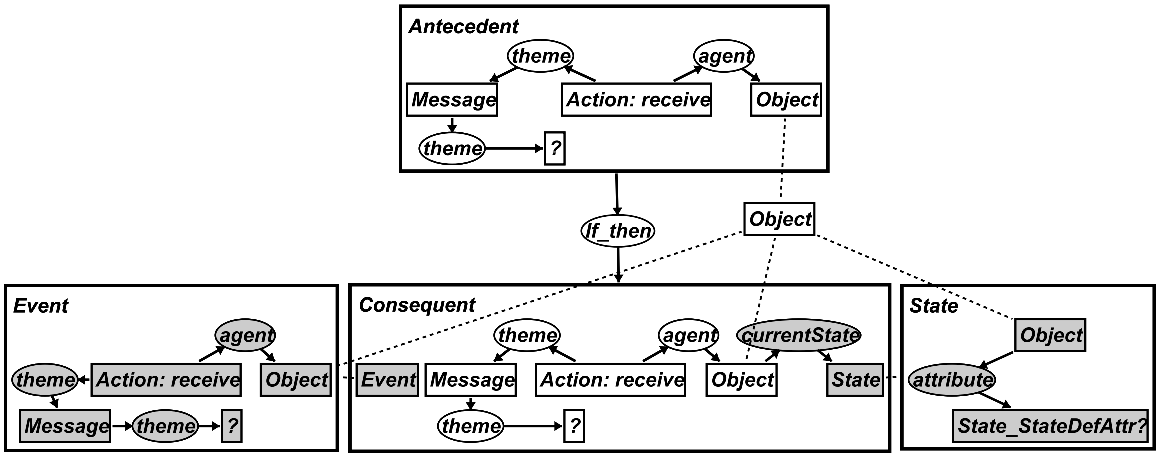


Figure 3.7: Event inference rule for state diagrams

In this work, for each type of UML diagram, a set of CGs inference rules is defined in order to derive requirements knowledge from the CGs Reservoir to build UML diagrams. These are explored further in Chapter 5. In this overview chapter, we present some examples of inference rules for class diagrams in English rather than in CGs, such as:

- An *Object* type concept with an *attribute*, *operation*, or *association* relation implies a candidate class;
- Two *Object* type concepts communicating through a message imply a candidate association between two classes.

3.2.2 The CGs Reservoir

The CGs Reservoir stores UML diagrams in CGs form and is used to generate UML diagrams in phase 2. Other possible uses are discussed in Section 6.6.

3.3 Comparison of the Two Frameworks

The two frameworks in this work can be used to facilitate requirements knowledge acquisition. By carrying out model transformations, both are able to present a requirements modeler with generated analysis models with semantic holes which the modeler can start to fill in or modify. Since they have well-defined semantic descriptions and are based on the same semantic source (the standard UML semantics), a comparison is possible. In order to create this comparison, we apply the two frameworks to three non-trivial case studies, each from a different application domain. The three case studies are University Information System (UnivSys.) which is an information system, Cryptanalysis System (Cryptanlys.) which is an AI system, and Mine Safety Control System (MineSys.) which is a real time control system. These are taken directly from the following books with no or minor modifications: *The Object Primer: Agile Model-Driven Development with UML 2.0* [29], *Object-oriented analysis and design with applications* [30], and *Requirements Engineering: From System Goals to UML Models to Software Specifications* [31].

The way to conduct our experiment is shown in Figure 3.8 to Figure 3.10. In Figure 3.8, we assume that the class diagram is not available, so state and sequence diagrams are used to generate class diagrams with semantic holes in both frameworks;

in Figure 3.9, the state diagram is assumed not available, so class and sequence diagrams are used to generate state diagrams with semantic holes in both frameworks; in Figure 3.10, the assumption is that the sequence diagram is not available, so we can use class and state diagrams to generate sequence diagrams with semantic holes in both frameworks. The results of working out the three case studies in both frameworks are very lengthy (about 50 pages each), so they are put in the appendices of this dissertation. Their structures are identical. Readers may wait to look at the appendices when they read through Chapter 5. Based on the results in the appendices, we can compare the two frameworks.

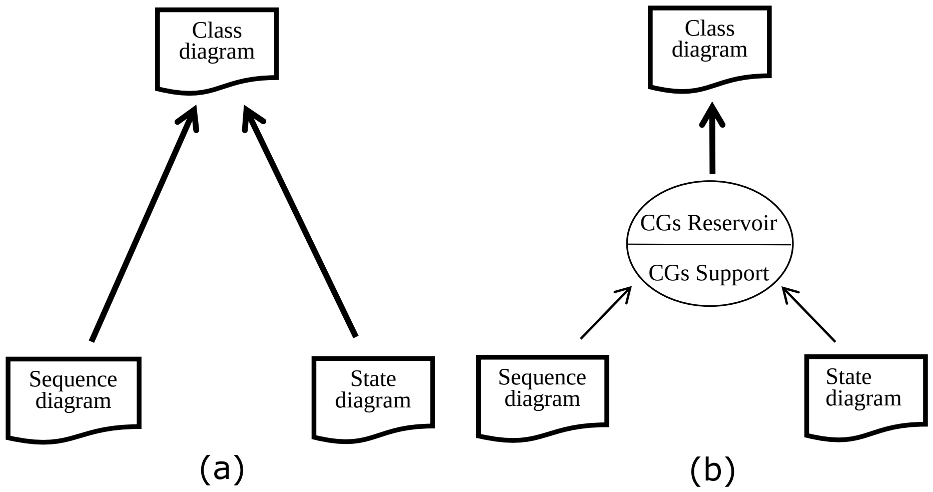


Figure 3.8: The experiment part 1

We propose a set of criteria that can be used to evaluate frameworks used for requirements knowledge acquisition, thereby compare the two frameworks in our work. These criteria are not limited to the two frameworks developed in this dissertation; they are meant to be applied to any framework that claims to address

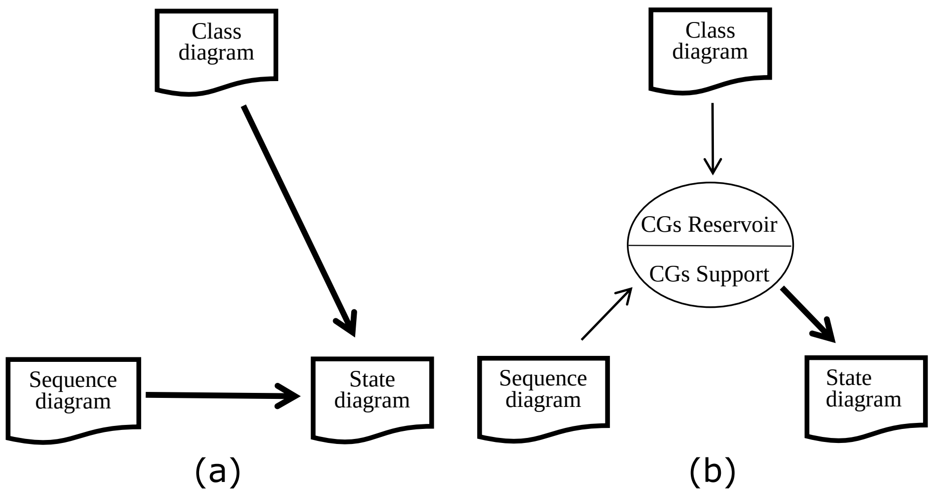


Figure 3.9: The experiment part 2

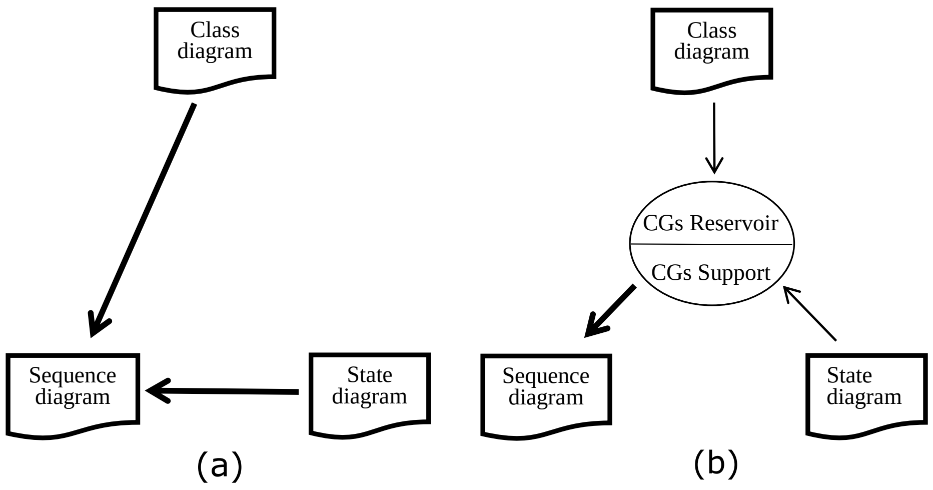


Figure 3.10: The experiment part 3

the requirements knowledge acquisition problem. The criteria are divided into two categories: quantitative and qualitative. The set of criteria is listed in Table 3.1.

The first two criteria are concerned with the size of the generated UML diagrams. For both frameworks, a generated UML diagram consists of two parts: the missing requirements represented as semantic holes which need to be clarified by

Table 3.1: Criteria for evaluating requirements acquisition frameworks

NO.	Criterion name	Criterion description
		Quantitative criteria
1	Capability of acquiring missing requirements	This criterion measures the number of the missing requirements that can be potentially acquired from a modeler given a generated UML diagram.
2	Capability of generating definite requirements	This criterion measures the number of the definite requirements that is generated in a generated UML diagram from other existing UML diagrams.
3	Percentage of the missing requirements in generated UML diagrams	This criterion represents the percentage of the missing requirements in a generated UML diagram.
4	Extensibility	This criterion evaluates the ability to include a new type of UML diagram in the framework.
5	Knowledge acquisition effort	This criterion measures the effort of eliciting knowledge from requirements modelers.
		Qualitative criteria
6	Capability of reasoning in requirements knowledge	This criterion determines whether or not the framework provides reasoning capability or not.

modelers and the definite requirements which are generated correctly and do not need clarification. Criterion 1 measures the number of semantic holes generated and criterion 2 measures the number of definite requirements generated.

Criterion 3 calculates the percentage of semantic holes in a generated UML diagram. In other words, it measures how incomplete a UML diagram generated by a framework is. Criterion 4 evaluates the extensibility of a framework. Criterion 5 measures the amount of effort it requires to complete a UML diagram with semantic holes generated by a framework.

For a complete description, evaluation and comparison of the two frameworks based on results of the three case studies, see Chapter 6.

CHAPTER 4

PAIR-WISE FRAMEWORK

In this chapter, pair-wise transformation rules for the three UML diagrams in PWF ((a) of Figure 3.1) are developed and presented. The transformation process in the framework only involves the transformation between two models:

$$Model_{source} \rightarrow Model_{target}$$

4.1 From Sequence Diagrams to Class Diagrams

A sequence diagram arranges a set of messages passed between objects in a timely manner [32]. The objects represented by lifelines should be defined in class diagrams, and messages between objects are treated as operations or receptions. Detailed transformation rules are summarized in Table 4.1.

4.2 From State Diagrams to Class Diagrams

A state diagram specifies the sequences of states an object goes through during its lifetime in response to events together with its responses to those events [33]. The class of the object that a state diagram describes should be defined in class diagrams and the events received by the state machine actually come from objects which are instances of neighboring classes. Detailed transformation rules are summarized in Table 4.2.

Table 4.1: Rules for transforming sequence diagrams to class diagrams

Model element(s) in sequence diagrams	**Corresponding element(s) in class diagrams**
Lifelines	Class definitions
Synchronous messages	Operations of the receiver object's class
Asynchronous messages	Receptions of the receiver object's class
Arguments of synchronous message	Parameters of the operation and the attributes of the sender object's class
Arguments of asynchronous message	Attributes of a signal class and attributes of the sender object's class
Two communicating lifelines	Association between two classes
The direction of a message	Association direction

Table 4.2: Rules for transforming state diagrams to class diagrams

Model element(s) in state diagrams	**Corresponding element(s) in class diagrams**
State machines	Class definitions
States	Values of an attribute called State
Call events	Operations of the class
Signal events	Receptions of the class
Change events	Values of an attribute of a neighboring class or current class
Entry, exit, do activities and effects	Operations of the class
Transitions	Operations named in the form “SourceStateToTargetState()” of the class
Guards	Values of an attribute of a neighboring class or current class

4.3 From Sequence Diagrams to State Diagrams

Transforming sequence diagrams to state diagrams is a classic problem. In this work, we adopted Grønmo and Møller-Pedersen's approach [34] to transform sequence diagrams to state diagrams. According to [34], each lifeline corresponds to a state machine. So when producing a state machine, it is sufficient to look at the single corresponding lifeline with its messages. An incoming arrow triggers a state transition while an outgoing arrow is treated as the effect on the transition. Detailed transformation rules in [34] are summarized in Table 4.3.

Table 4.3: Rules for transforming sequence diagrams to state diagrams

Model element(s) in sequence diagrams	**Corresponding element(s) in state diagrams**
Lifelines	State machines
Incoming synchronous messages	Call events that trigger state transitions
Return messages	Return events that trigger state transitions
Incoming asynchronous messages	Signal events that trigger state transitions
Outgoing synchronous messages	Effects on the state transition (calling other object's method)
Outgoing asynchronous messages	Effects on the state transition (a send action)

4.4 From Class Diagrams to State Diagrams

A state diagram describes the life cycle of an object during the execution of the system. Even though the states of a state diagram can be related to values of attributes of a class in class diagrams, it is impossible to infer the behaviors of an object given just the structural information. As such, we are not able to develop rules to transform class diagrams to state diagrams.

4.5 From Class Diagrams to Sequence Diagrams

Similar to transforming class diagrams to state diagrams, it is impossible to infer the sequential knowledge needed among objects given just the structural information. Because of this, we are not able to develop rules to transform class diagrams to sequence diagrams.

4.6 From State Diagrams to Sequence Diagrams

The algorithms of transforming state diagrams to sequence diagrams come from our unpublished paper [35]. Transformation rules of model elements in state diagrams to model elements in sequence diagrams are summarized in Table 4.3.

Table 4.4: Rules for transforming state diagrams to sequence diagrams

Model element(s) in state diagrams	**Corresponding element(s) in sequence diagrams**
State machines	Lifelines
States	Annotated state information between message passing along the lifeline
Call events	Synchronous messages
Signal events	Asynchronous messages
Change events	Combined fragments
Entry, exit, do activities and effects	Execution Specifications
Guards	Combined fragments

However, deriving knowledge of “sequences” from a state diagram is tricky. Since a state diagram specifies sequences of states an object goes through during its lifetime, this would result in multiple sequence diagrams, each of which describe one possible state transition path. Nevertheless, it is impractical to generate and present all of them to modelers. For the purpose of knowledge acquisition, the set of generated sequence diagrams from a state diagram possesses the following **five properties** that can be verified in this work:

1. Each sequence diagram in the set starts with the initial state (the object is created) and ends in a final state (the object is destroyed);

2. Every state transition in the state diagram is covered at least once in the set of generated sequence diagrams;

3. The number of sequence diagrams is minimal given that property 2 is satisfied;

4. Each sequence diagram in the set has only one lifeline which represents the object owning the state diagram;

5. There are knowledge acquisition opportunities so the modeler is given a chance to fill in the missing requirements.

The above list summarizes the properties of the set of generated sequence diagrams in this work. Property 1 makes sure that each of the generated sequence diagrams conveys a complete scenario to the sequence diagram modeler. Property 2 and 3 guarantee that the modeler is presented with UML messages implied by all the state transitions in state diagram; meanwhile the set of sequence diagrams is minimal, so it will not overwhelm the modeler. Since one state diagram only captures the behavior of one object, property 4 says each sequence diagram only has one lifeline. Property 5 shows the modeler knowledge acquisition opportunities in the form of meaningful labels so that she can start filling in semantic "holes" with the missing requirements knowledge.

Detailed algorithms for generating a minimal set of sequence diagrams that satisfy the five properties can be found in our unpublished paper [35].

4.7 Summary

In this chapter, transformation rules between each pair of the three UML diagrams are specified. Using these rules, a target UML diagram can be augmented or generated from scratch by transforming the other two UML diagrams into the target UML diagram. During the transformation, semantic holes are generated and can be used as a way to make the target UML diagram more complete. At this point, readers are able to look at the PWF portion of the three complete case studies in the appendices, they are Section A.1, Section B.1 and Section C.1 in the appendices. As readers can see, the PWF portion of each case study is much shorter than the CRF portion.

CHAPTER 5

COMMON REPRESENTATION FRAMEWORK

In Chapter 3, we gave an overview of our CRF and introduced a key component, the CGs Support. In this chapter, we elaborate on the model transformations in CRF. Section 5.1 shows how three types of UML diagrams are converted to CGs; generating UML diagrams from the CGs Reservoir is presented in Section 5.2.

5.1 Converting UML Diagrams to CGs

Each type of UML diagram has a set of basic model elements upon which the diagram is built. For example, class diagrams include a set of basic model elements such as classes, attributes, operations and associations. In CRF, for each type of UML diagram, a set of canonical graphs is developed in which the basic model elements are properly represented in terms of the primitive concepts and relations. By instantiating the set of canonical graphs, the requirements knowledge of a UML diagram is represented in CGs. In this way, a UML diagram is converted to CGs.

In this section, canonical graphs for class diagrams, state diagrams and sequence diagrams are developed and presented in Section 5.1.1, Section 5.1.2 and Section 5.1.3, respectively; the conversion process of the three types of UML diagrams to CGs is also demonstrated.

5.1.1 Converting Class Diagrams to CGs

A class diagram describes the static view of a system in terms of collection of declarative classes and relationships. For simplicity, the basic model elements of class diagrams we considered in this work are classes, attributes, operations and associations.

5.1.1.1 Canonical Graphs for Class Diagrams

The canonical graph for a class is shown in (a) of Figure 5.1. Since a class describes a set of similar objects, the concept *ClassName: @forall* means "for all objects of this class." In this concept, *ClassName* denotes the name of a class and will be replaced with the name of a real class when this canonical graph is instantiated. An attribute is represented by a T type concept, which is related to the class concept through *attribute* relation, while an operation is represented by an *Activity* type concept, which is related to the class concept by *operation* relation. The *association* relates each object of this class to other objects. In (a) of Figure 5.1, only one attribute, one operation, and one association are shown in the canonical graph. Canonical graphs (b) and (c) of Figure 5.1 represent composition and generalization, respectively.

The canonical graph (a) of Figure 5.1 represents "For each object of a class ClassName (ClassName is used as a placeholder), it has an attribute of type T, an

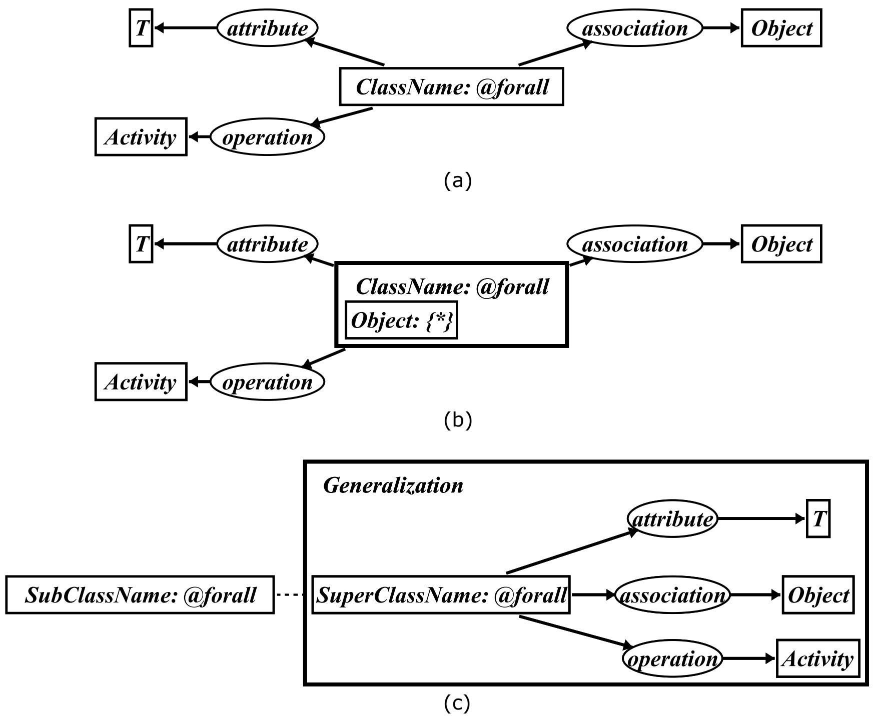


Figure 5.1: Canonical graphs for class diagrams

operation of type Activity and is associated with an object of type Object." Note that, concept types *T*, *Activity* and *Object* are primitive concept types in the primitive concept type hierarchy (Figure 3.3), and relation types *attribute*, *operation* and *association* are primitive relations defined in the primitive relation type hierarchy (Figure 3.4).

5.1.1.2 Example of Class Diagrams in CGs

The class diagram of the UnivSys. is shown in Figure 5.2.

By instantiating the canonical graph for class diagrams (in this case, (a) of Figure 5.1), the class diagram is converted to the CGs in Figure 5.3. Note that both class diagram and its corresponding CGs are colored orange.

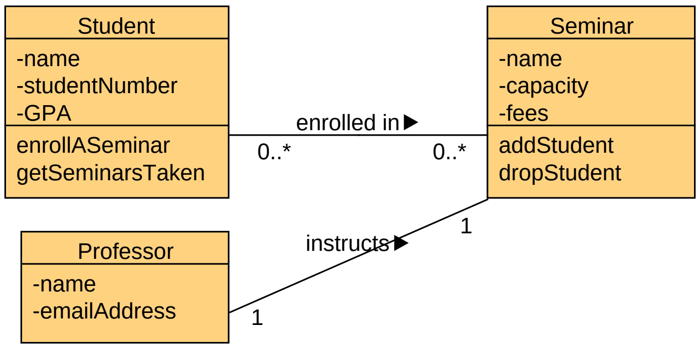


Figure 5.2: Class diagram of the UnivSys.

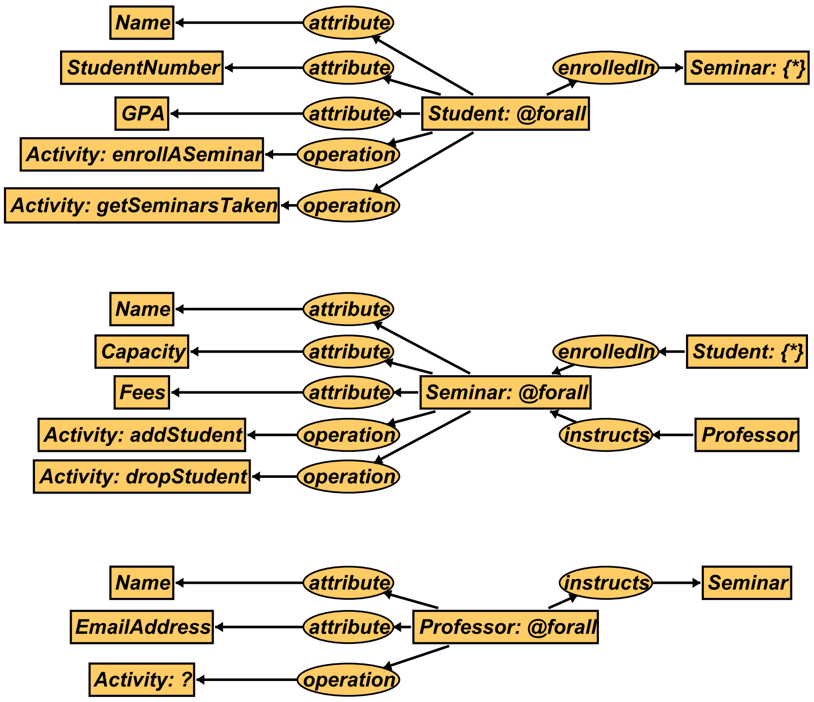


Figure 5.3: Class diagram of the UnivSys. in CGs

An interpretation of the first CG in Figure 5.3 is "For each Student object, it has Name, StudentNumber and GPA as its attributes, enrollASeminar and getSeminarsTaken as its operations, and is associated with a set of Seminar objects."

5.1.2 Converting State Diagrams to CGs

A state diagram consists of states connected by transitions. The occurrence of an event may fire a transition that causes the object to exit the current state and enter a new state. When a transition fires, an effect attached to the transition is performed. State diagrams aim at capturing all admissible sequences of state transitions. The basic model elements of state diagrams we considered in this work are states, events, guards, activities and transitions.

5.1.2.1 Canonical Graphs for State Diagrams

In CRF, a state diagram is viewed as a set of state transitions. Based on the semantic descriptions of state transitions, the canonical graph for one state transition is shown in Figure 5.4 (We have shown the canonical graph of a state transition in Chapter 3 and reprint it here).

The canonical graph in Figure 5.4 represents "An object is in a state performing an activity while an event occurs at time t1; the guard condition is also satisfied, so the object performs exit activity at time t2 and performs effects on the transition at time t3; before it enters the second state at time t5, the object performs the entry activity of the second state at time t4."

In the canonical graph of a state transition, object owning the state machine, effects on transitions, entry/exit and do activities of states are already represented by primitive concepts such as *Object* and *Activity*. However, basic model elements like events, states, and guards are still not expressed in primitive forms (see *Event*,

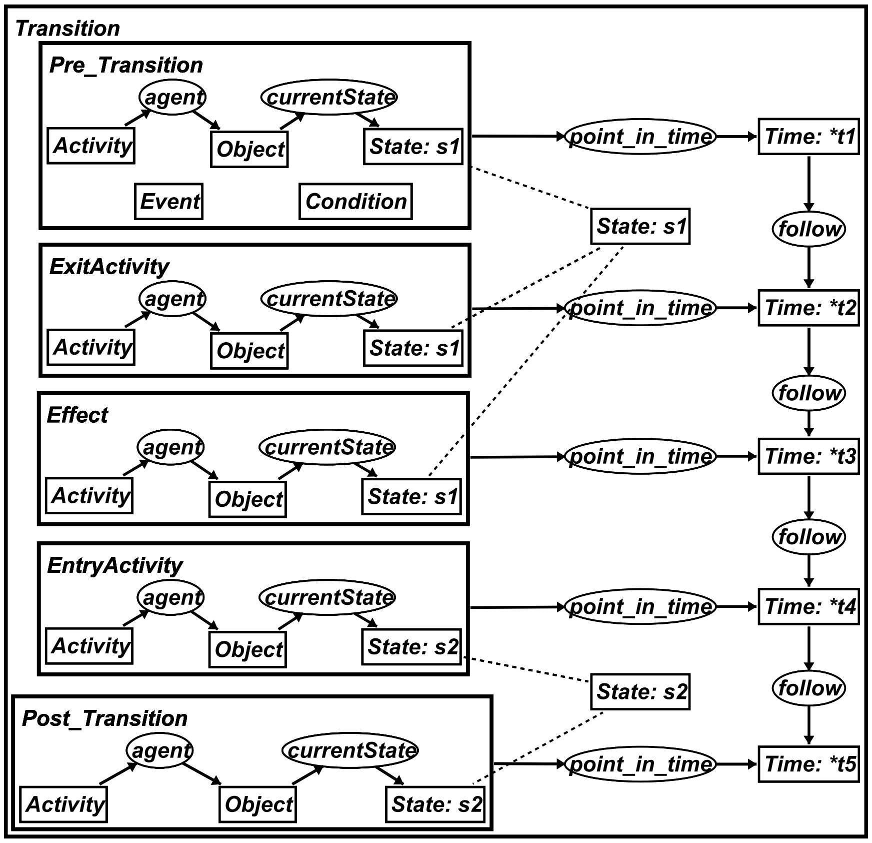


Figure 5.4: Canonical graph for a state transition in state diagrams

State and *Condition* concepts in Figure 5.4). More canonical graphs for those model elements are provided in Figure 5.5 to Figure 5.7.

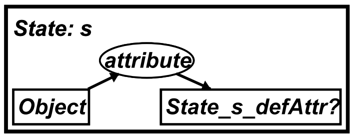


Figure 5.5: Canonical graph for states in state diagrams

In state diagrams, a state describes a situation during the life of an object during which it satisfies some condition, performs some do activity, or waits for some event [32]. Performing some do activity and waiting for some event are already defined in the *Pre_ Transition* context in Figure 5.4; satisfying some condition means that the state variable which defines a state always has the same value regardless of other state variables [31]. State variables are actually attributes of the object, so we come up with the definition of a state in CGs (see Figure 5.5). For example, in a train control system, when a train is in state `DoorsClosed`, the state variable `doorStatus` that defines that state must be equal to `closed`. However, the knowledge of state variable that defines a state is usually not available in a state diagram, so we make the "unavailability" explicit in CGs. Such incompleteness is a perfect example of knowledge acquisition opportunity. A state diagram modeler is encouraged to fill it in.

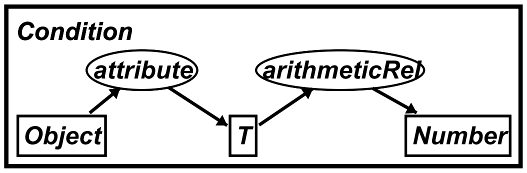


Figure 5.6: Canonical graph for guards in state diagrams

A guard condition is a Boolean expression that is part of a transition. For simplicity, we only consider Boolean expressions which involve arithmetic relational evaluations (Figure 5.6).

An event is something that happens during execution of a system that is worth modeling [32]. In a state diagram, a trigger on a state transition specifies an event whose occurrence enables the transition. There are four kinds of events that can be used in triggers [32]: signal event which is the receipt of a signal ((a) of Figure 5.7); call event which is the receipt of a call ((b) of Figure 5.7); change event which is the satisfaction of a Boolean condition specified by an expression in the event ((c) of Figure 5.7); and time event which is the satisfaction of a time expression, such as the occurrence of an absolute time or the passage of a given amount of time after an object enters a state ((d) of Figure 5.7).

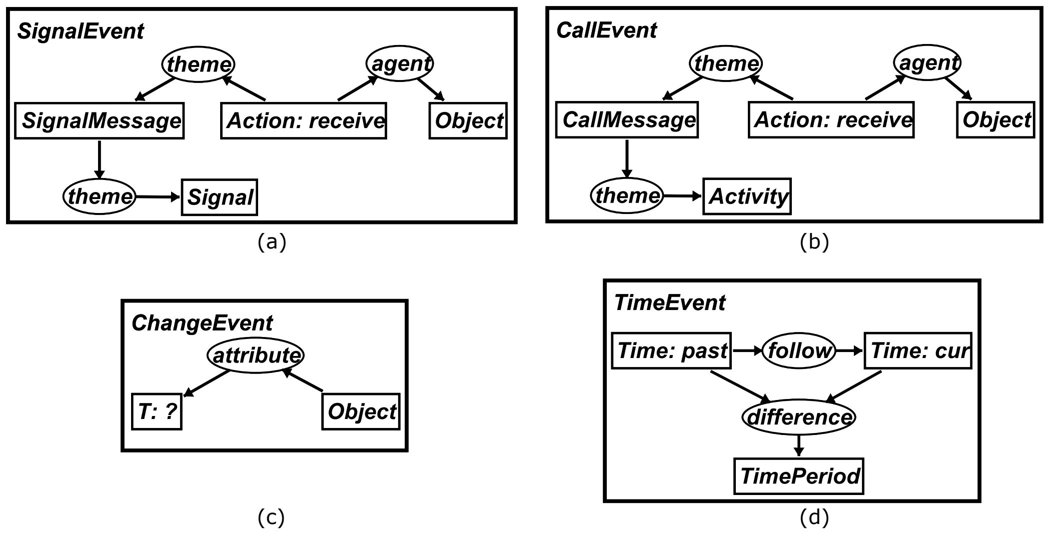


Figure 5.7: Canonical graphs for events in state diagrams

5.1.2.2 Example of State Diagrams in CGs

Based on the canonical graphs defined for state diagrams, one of the 14 state transitions in the state diagram of the UnivSys., the self-transition “Open For Enrollment to Open For Enrollment” in Figure 5.8 is represented as CGs in Figure 5.9. Note that both state diagram and its corresponding CGs are colored green.

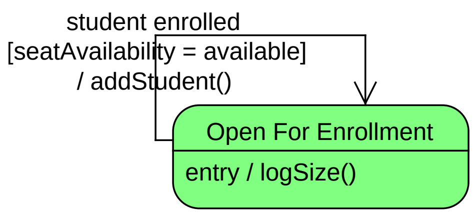


Figure 5.8: A snippet of the state diagram of the UnivSys.

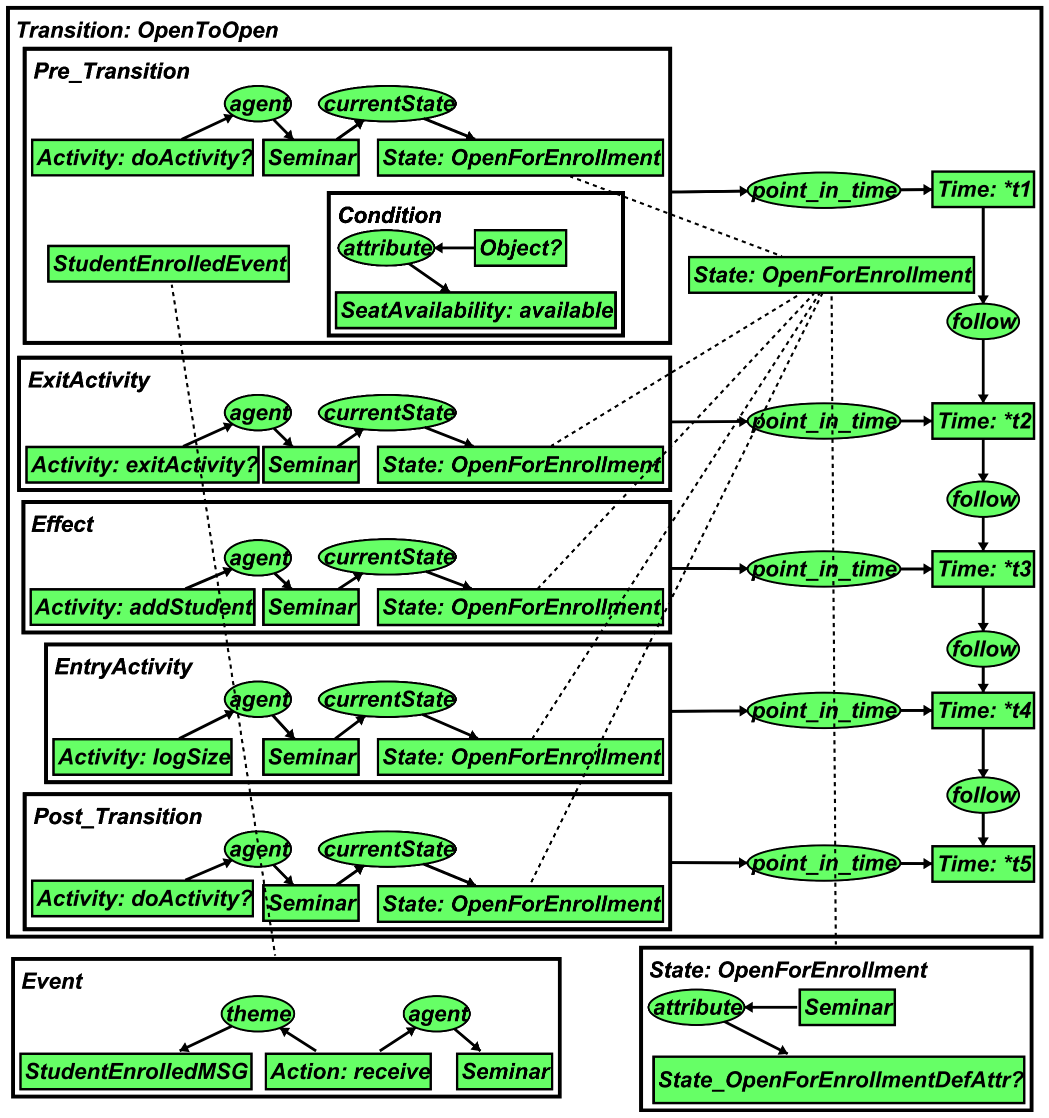


Figure 5.9: State transition "Open to Open" in CGs

Note how complicated one state transition in a state diagram is; the CGs in Figure 5.9 reflect the rich semantics of one state transition. Complete CGs of the entire state diagram of the UnivSys. can be found in Appendix A.

5.1.3 Converting Sequence Diagrams to CGs

A sequence diagram specifies an interaction where a set of messages are exchanged between objects in a time order [11]. Basic model elements of sequence diagrams include lifelines, messages, and execution specifications. We don't consider combined fragments like loop and alt (see Section 7.2).

5.1.3.1 Canonical Graphs for Sequence Diagrams

Based on the semantics of a sequence diagram, its CGs are composed of a sequence of *MessagePassing* contexts (Figure 5.10).

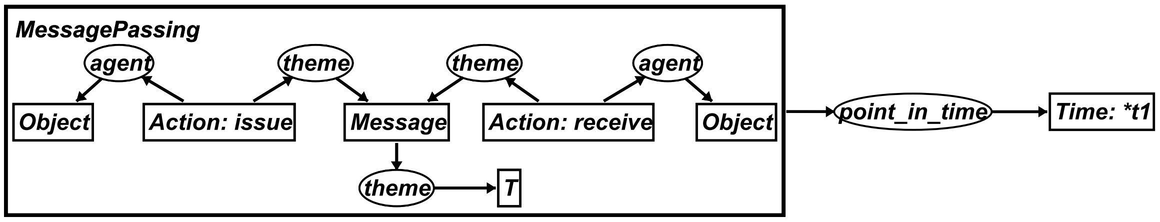


Figure 5.10: A message passed between two lifelines in CGs

The canonical graph in Figure 5.10 describes the semantics of a message exchange between objects at a certain point in time; a *Message* can carry parameters. In CRF, messages are further categorized into call messages and signal messages, i.e., synchronous and asynchronous messages, respectively, so the canonical graph in Figure 5.10 is extended to two more specific canonical graphs (Figure 5.11 and Figure 5.12).

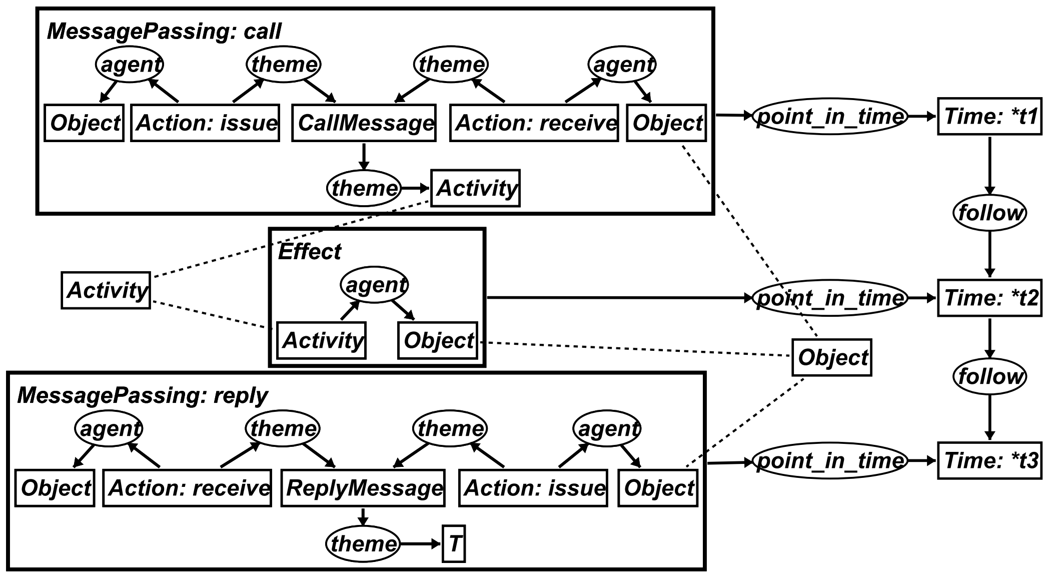


Figure 5.11: Canonical graph for a call message passing

The canonical graph in Figure 5.11 represents "At time t1, one lifeline sends a synchronous call message to another lifeline which carries out the operation specified in the message at time t2 and returns T to the sender at time t3." Note that all the concepts and relations in Figure 5.11 are already primitives.

The canonical graph in Figure 5.12 represents "At time t1, one lifeline sends an asynchronous signal message to another lifeline which processes the signal at time t2." Since a signal message is asynchronous, there is no return message. Note that all the concepts and relations in Figure 5.12 are already primitives.

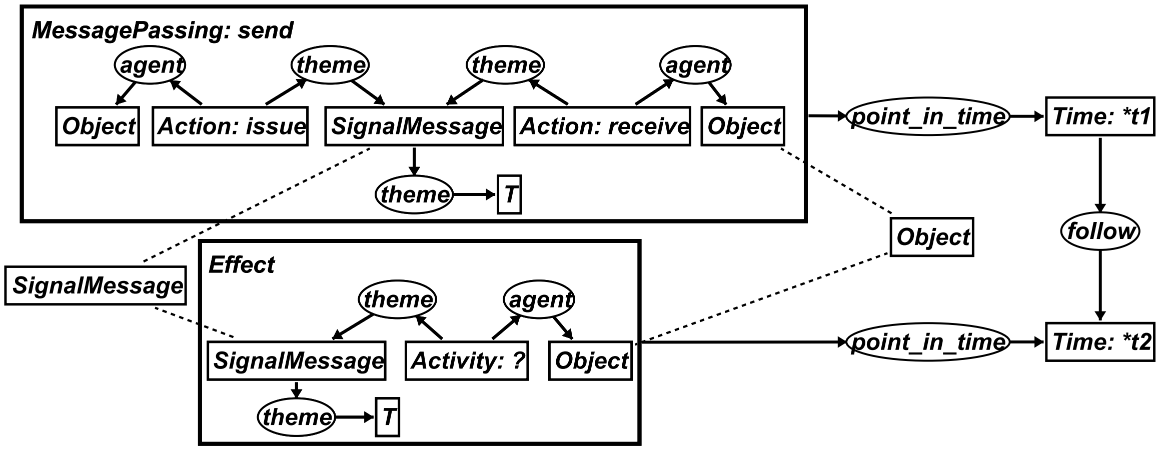


Figure 5.12: Canonical graph for a signal message passing

5.1.3.2 Example of Sequence Diagrams in CGs

In the UnivSys. case study, when a student wants to enroll in a seminar, the student's qualification must be determined first. In Figure 5.13, we take a snippet of the original sequence diagram in [4] to form a simple sequence diagram.

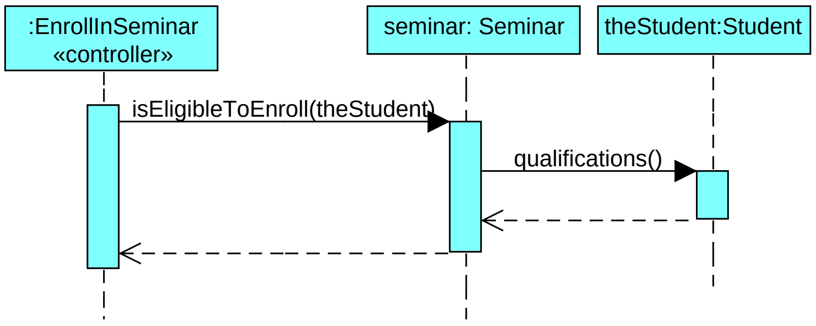


Figure 5.13: A snippet of the sequence diagram of the UnivSys.

By instantiating the canonical graph for sequence diagrams, the sequence diagram in Figure 5.13 is converted to CGs in Figure 5.14. Note that both sequence diagram and its corresponding CGs are colored cyan.

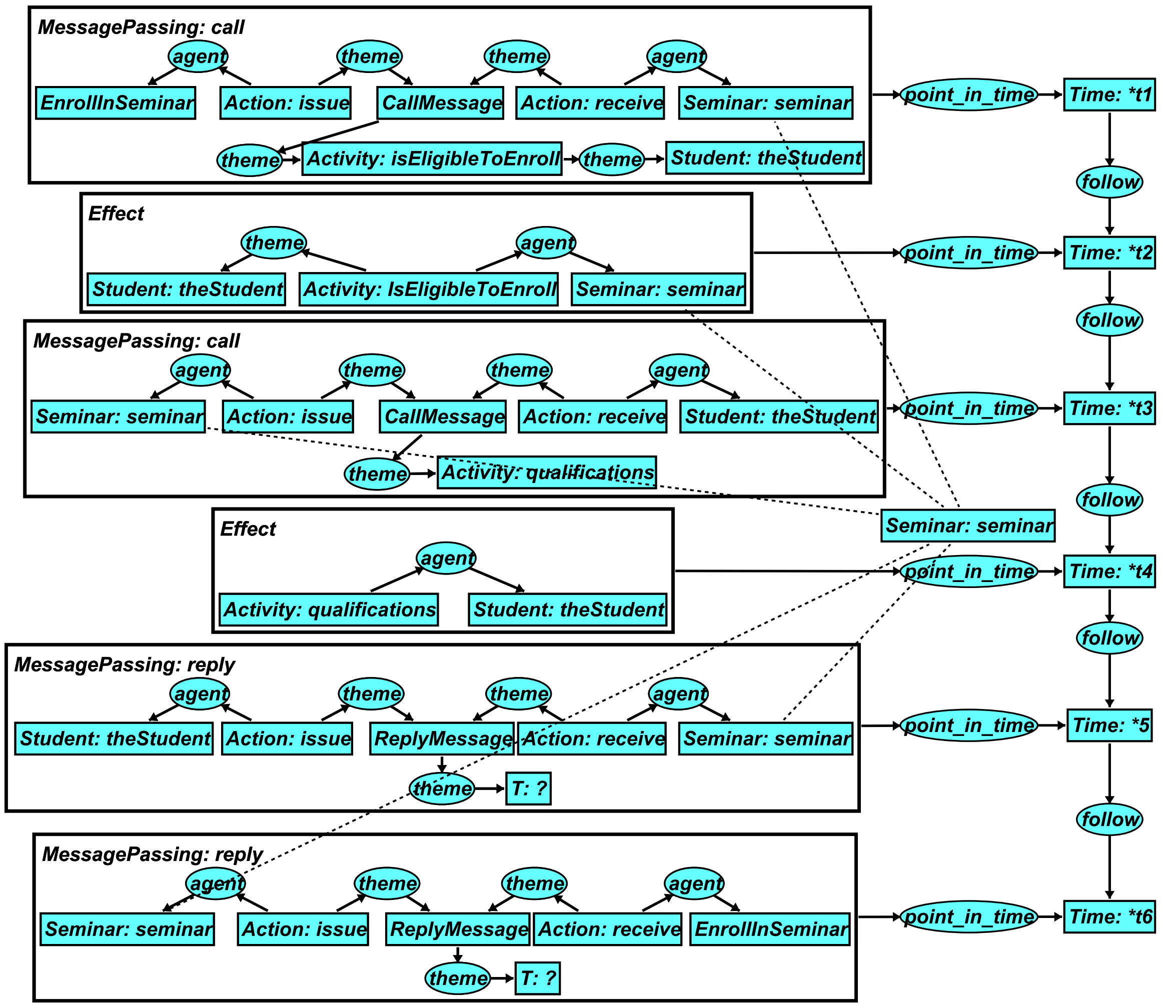


Figure 5.14: A snippet of the sequence diagram of the UnivSys. in CGs

5.2 Generating UML Diagrams with Semantic Holes from the CGs Reservoir

In Section 5.1, we showed how class, state and sequence diagrams are converted to CGs so that the CGs Reservoir can be populated. In this section, the CGs Reservoir containing requirements knowledge is used to generate UML diagrams. The generated UML diagrams contain semantic holes which provide knowledge acquisition opportunities so that eliciting new requirements knowledge becomes possible.

The processes of generating class diagrams with semantic holes, state diagrams with semantic holes and sequence diagrams with semantic holes are elaborated upon in Section 5.2.1, Section 5.2.2 and Section 5.2.3, respectively.

5.2.1 Generating Class Diagrams from the CGs Reservoir

In this subsection, the CGs Reservoir is used to generate class diagrams. The CGs Reservoir is assumed to include the requirements knowledge of state and sequence diagrams expressed in CGs already.

5.2.1.1 Inference Rules for Class Diagrams

Inference rules for inferring requirements knowledge necessary for building class diagrams are developed in Figure 5.15 and Figure 5.16. In these figures, CGs in light gray denote the inferred requirements knowledge.

The class-level inference rules in Figure 5.15 are detailed here:

- Class rule: If an *Object* type concept appears, then its class needs to be defined;
- Attribute rule: If an *Object* type concept is related to a concept *y through *attribute*, then the concept *y is an attribute value of the class of the *Object* type concept;
- Operation rule: If an *Object* type concept performs a *Process* type concept *y, then concept *y is an operation of the class of the *Object* type concept;
- Association rule: If two *Object* type concepts have the *association* relation between them, then their classes are associated.

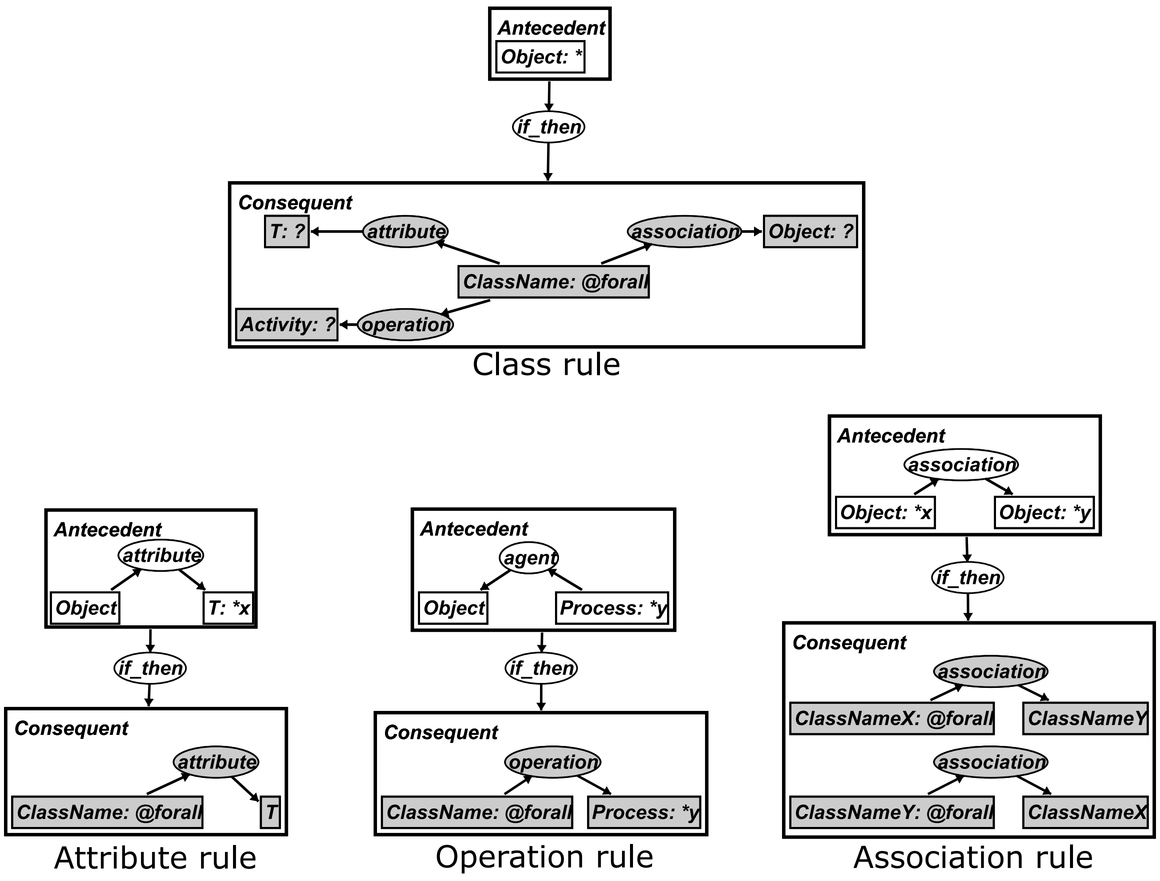


Figure 5.15: Class diagram inference rule set 1

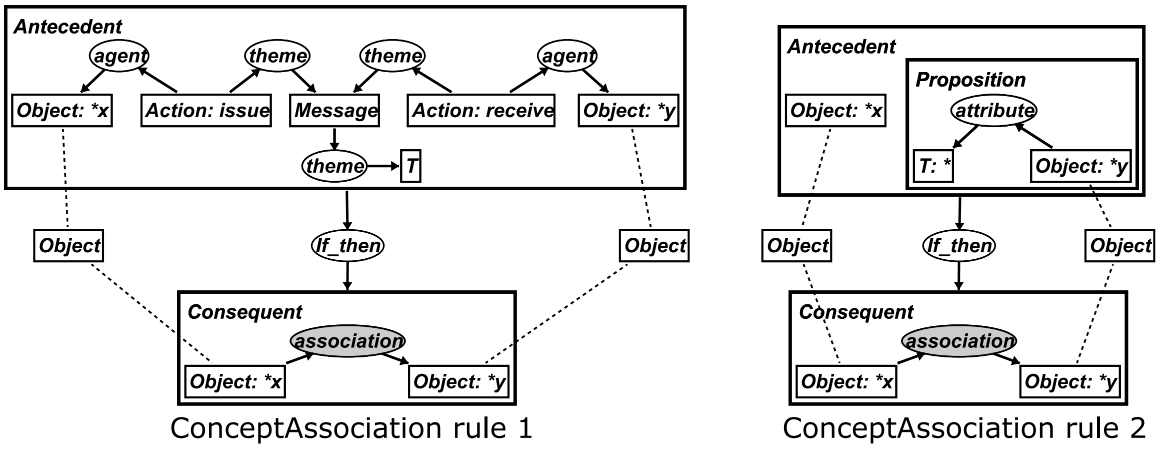


Figure 5.16: Class diagram inference rule set 2

For the class-level Association rule in Figure 5.15, in order to infer an *association* relation between two concepts, two more inference rules at object-level are needed (Figure 5.16):

- ConceptAssociation rule 1: If two *Object* type concepts communicate through passing messages, then they have the *association* relation between them;
- ConceptAssociation rule 2: If one *Object* type concept can access the attribute of another, then they have the *association* relation between them.

5.2.1.2 Inferring Class Diagrams from CGs

We will now demonstrate how to use class diagram inference rules to infer requirements knowledge for building class diagrams. The CGs Reservoir already contains requirements knowledge of the state and the sequence diagrams of the MineSys. case study. For simplicity, only one inferred class is shown in this subsection. See Appendix C for the complete generated class diagram of Mine Safety Control System.

ConceptAssociation rule 1 and ConceptAssociation rule 2 are first applied to the current CGs Reservoir to look for associations among objects. Then we apply the first rule set to infer the class CGs model.

In Figure 5.17, for class *HighWaterSensor*, three attributes and nine operations are inferred from the CGs Reservoir; two neighboring classes are also inferred.

The generated CGs in Figure 5.17 are then translated back to UML class diagram notations (Figure 5.18). Besides some definite requirements just mentioned, several semantic holes are generated: attributes and operations in class `Sump` and

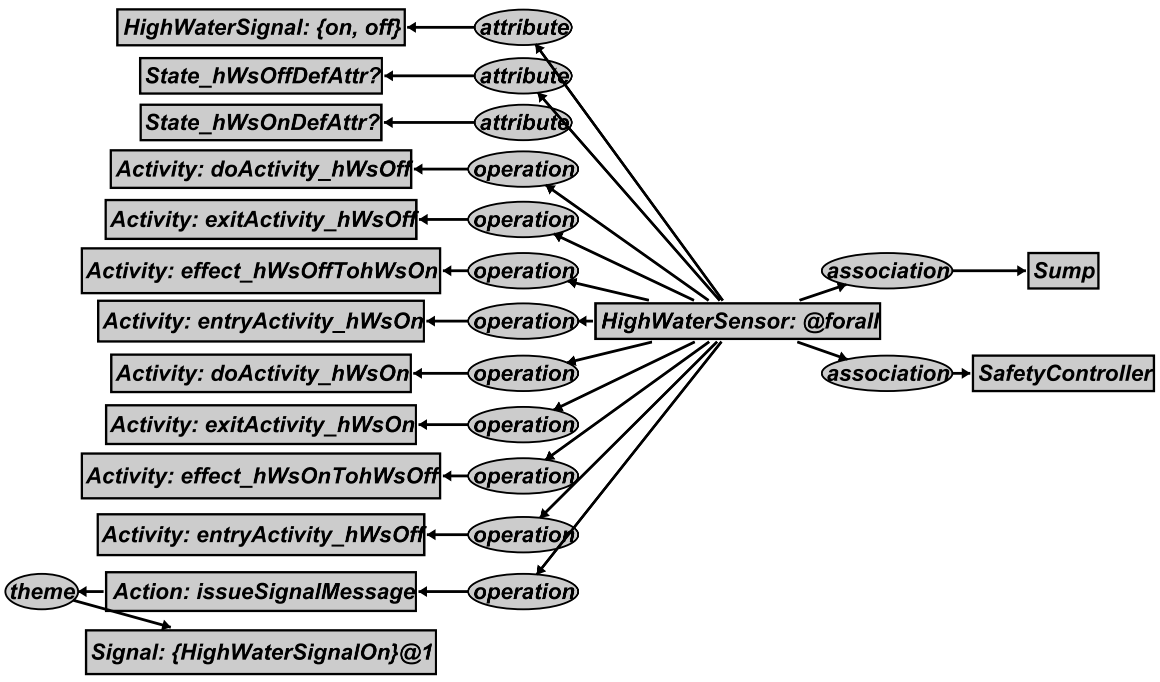

HighWaterSignal: {on, off}
attribute
State_hWsOffDefAttr?
attribute
State_hWsOnDefAttr?
attribute
Activity: doActivity_hWsOff
operation
Activity: exitActivity_hWsOff
operation
Activity: effect_hWsOffTohWsOn
operation
association
Sump
Activity: entryActivity_hWsOn
operation
HighWaterSensor: @forall
Activity: doActivity_hWsOn
operation
association
SafetyController
Activity: exitActivity_hWsOn
operation
Activity: effect_hWsOnTohWsOff
operation
Activity: entryActivity_hWsOff
operation
theme
Action: issueSignalMessage
operation
Signal: {HighWaterSignalOn}@1

Figure 5.17: Generated CGs class model after applying inference rules

`SafetyController`, and their associations with class `HighWaterSensor` need to be filled in; two attributes `State_hWsOffDefAttr` and `State_hWsOnDefAttr` in class `HighWaterSensor` need to be clarified; operations like `doActivity_hWsOff` need to be renamed.

This generated incomplete diagram clearly shows a knowledge acquisition opportunity: a class diagram modeler presented with the generated class diagram in Figure 5.18 can fill in the semantic holes by providing additional information. A possible class diagram after resolving semantic holes by a class diagram modeler is shown in Figure 5.19. Note that the two state variables inferred by two states `hWsOff` (high water signal off) and `hWsOn` (high water signal on) are considered unnecessary since the inferred attribute `HighWaterSignal` can be the state variable of both states. By

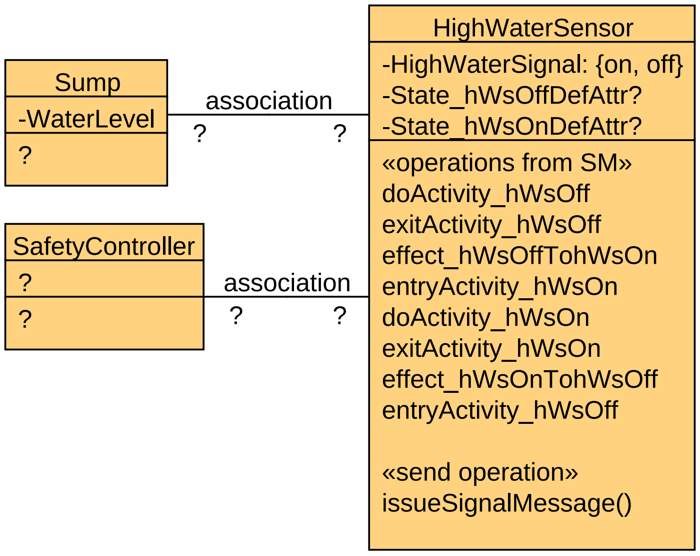


Figure 5.18: Generated class diagram with semantic holes

prompting the effects on transitions and activities in states, a class modeler provides the operations in class `HighWaterSensor`. Class `Sump` and `SafetyController` are also made more complete. The two associations are named; arities and navigations are made clear.

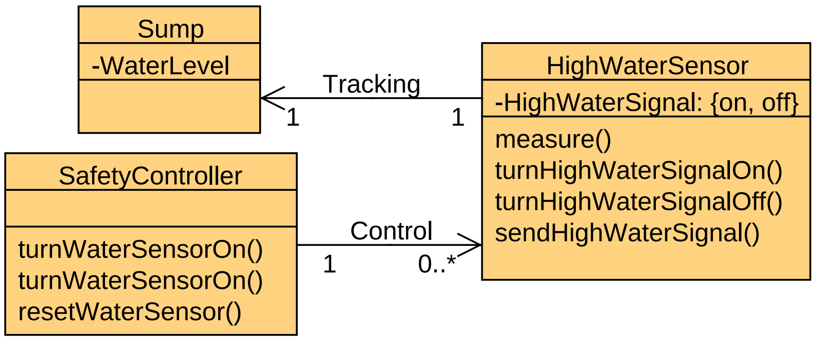


Figure 5.19: Class diagram after completion

Thus the generated incomplete model has fulfilled its purpose of inviting the modeler to provide additional requirements knowledge which will become a part of the collection of models for a given system being developed.

5.2.2 Generating State Diagrams from the CGs Reservoir

In this subsection, the CGs Reservoir is used to generate state diagrams. The CGs Reservoir is assumed to include the requirements knowledge of class and sequence diagrams expressed in CGs already.

5.2.2.1 Inference Rules for State Diagrams

Since events may trigger state transitions in a state diagram, the key to generating a state diagram is to look for events in the CGs Reservoir. The rule for inferring a message event is shown earlier in Figure 3.7 in Section 3.2.1.2. In a state diagram, there are several kinds of events: call events, signal events, change events, and time events. Both call and signal events are kinds of message events; their inference rules are shown in Figure 5.20 and Figure 5.21, respectively. However, our current CRF cannot support inferring change and time events.

The call event rule in Figure 5.20 represents "If in the CGs Reservoir, an object receives a call message, then this object is in a certain state and there exists a *CallEvent*; the state needs to be defined in terms of the attributes of the object." Note that the inferred concepts and relations are colored in light gray in Figure 5.20.

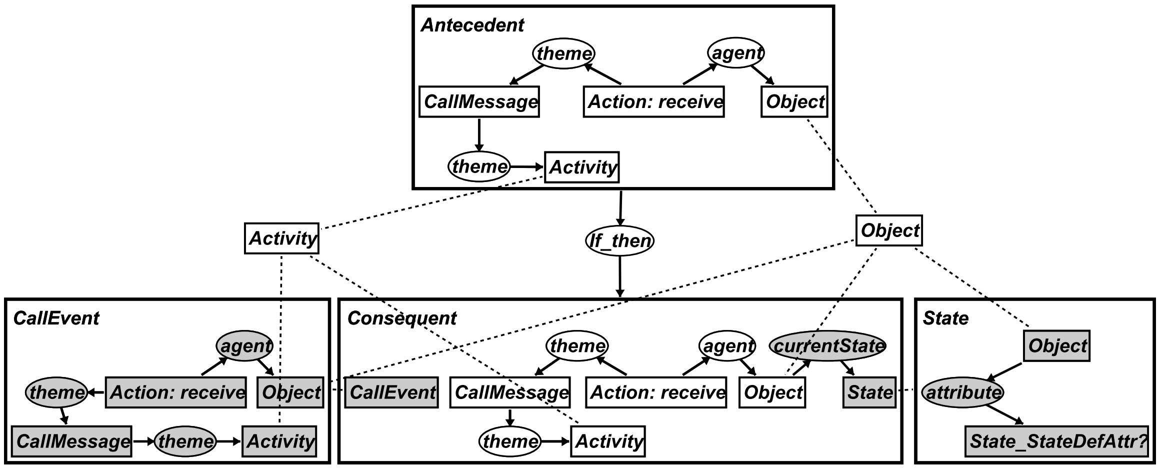


Figure 5.20: Call event rule

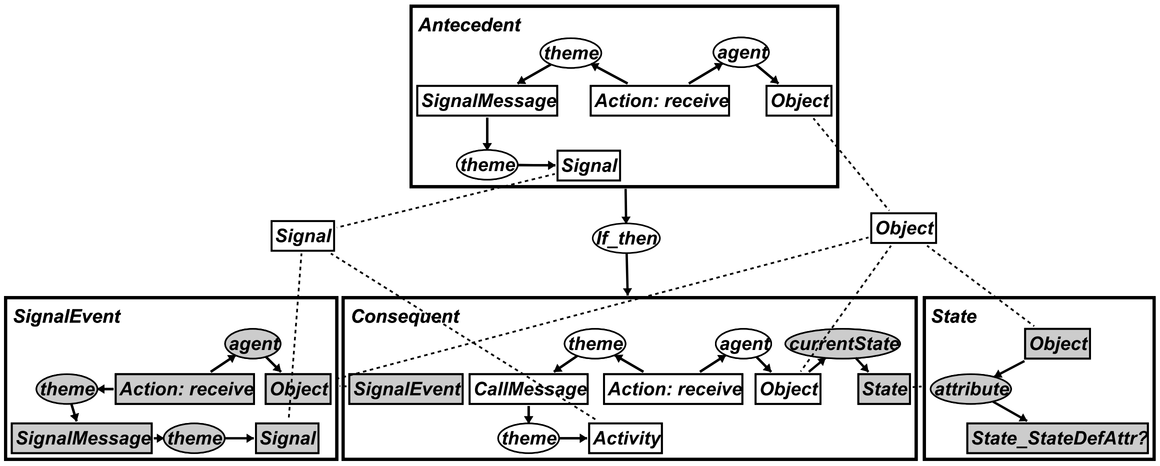


Figure 5.21: Signal event rule

The signal event rule in Figure 5.21 represents "If in the CGs Reservoir, an object receives a signal message, then this object is in a certain state and there exists a *SignalEvent*; the state needs to be defined in terms of the attributes of the object."

5.2.2.2 Inferring State Diagrams from CGs

We will now demonstrate how to use state diagram inference rules to infer requirements knowledge for building state diagrams. The CGs Reservoir contains requirements knowledge of the class and the sequence diagrams of the UnivSys. and we try to infer the state machine of the `Seminar` object. So the *Object* type concept in the call event inference rule in Figure 5.20 has been instantiated to *Seminar: seminar* (Figure 5.22).

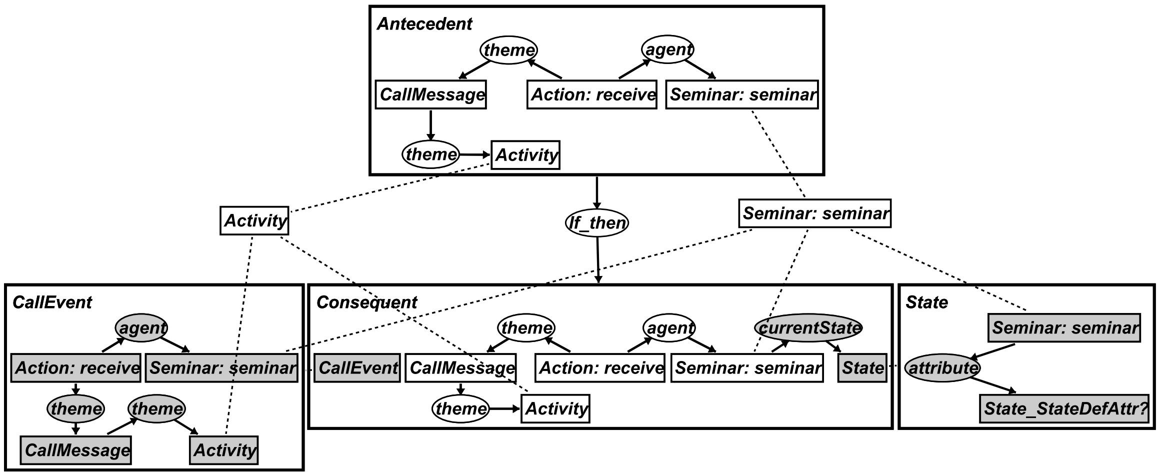


Figure 5.22: Instantiated call event inference rule

The instantiated inference rule is then applied to the current CGs Reservoir. New requirements knowledge (shown as gray concepts and relations in Figure 5.23) is added to the existing CGs in the CGs Reservoir (shown as cyan concepts and relations). We want to remind readers that cyan represents the requirements knowledge of sequence diagrams in the CGs Reservoir.

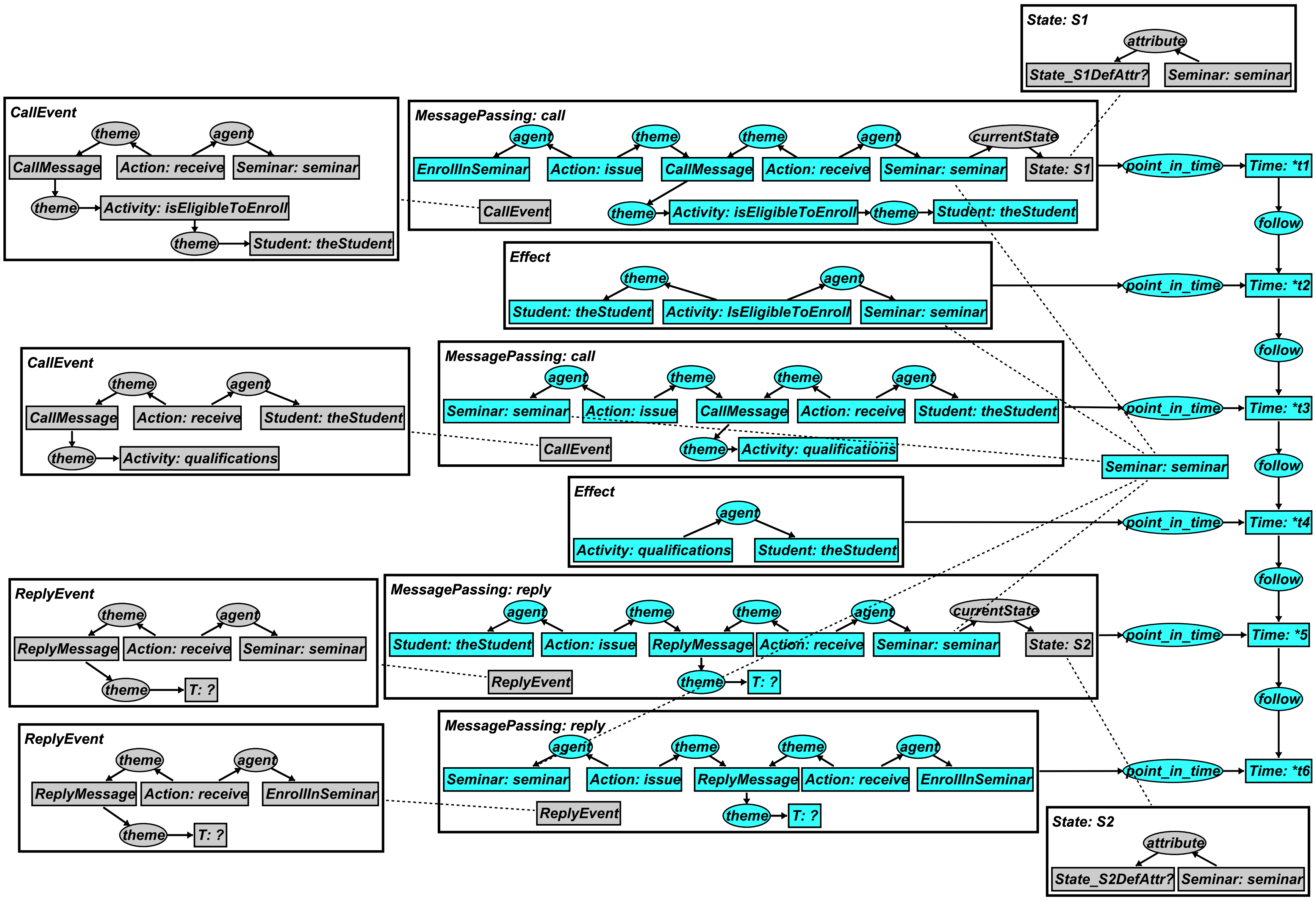


Figure 5.23: Augmented CGs model after applying the instantiated call event inference rule

Two pieces of state information, *S1* and *S2*, are inferred. When the CGs in Figure 5.23 are transformed back to the UML state diagram notations (Figure 5.24), state `S1` shows a transition to `S2` responding to the call event `isEligibleToEnroll`, and `S2` shows a transition to some unknown state that responds to the return value of the `qualifications` operation. Several semantic holes are thus identified: the missing definitions of state `S1` and `S2`, the entry/exit, and do activities of the two states, the conditions (guards) of the two transitions, and the target state of the transition from `S2`.

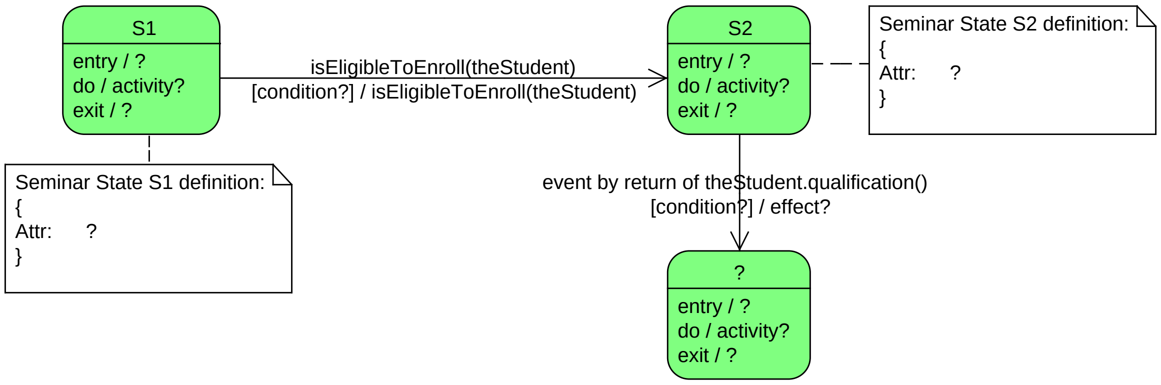

S1
entry / ?
do / activity?
exit / ?
isEligibleToEnroll(theStudent)
[condition?] / isEligibleToEnroll(theStudent)
S2
entry / ?
do / activity?
exit / ?
Seminar State S2 definition:
{
Attr: ?
}
Seminar State S1 definition:
{
Attr: ?
}
event by return of theStudent.qualification()
[condition?] / effect?
?
entry / ?
do / activity?
exit / ?

Figure 5.24: Generated state diagram with semantic holes

This generated incomplete diagram clearly shows a knowledge acquisition opportunity: a state modeler presented with the state model in Figure 5.24 can fill in the semantic holes by providing additional information. A possible state diagram after resolving semantic holes by a state diagram modeler is shown in Figure 5.25. Note that the unknown state with a question mark in Figure 5.24 is split into two states: `EnrollStudent` and `UnableToEnroll` based on the different return value of `qualification`. Also, a new attribute of the `Seminar` class `seatAvailability` is needed in order to define state `OpenForEnrollment`. When the more complete state diagram in Figure 5.25 is converted back to CGs, such knowledge will be used to augment the class model CGs for further inferences.

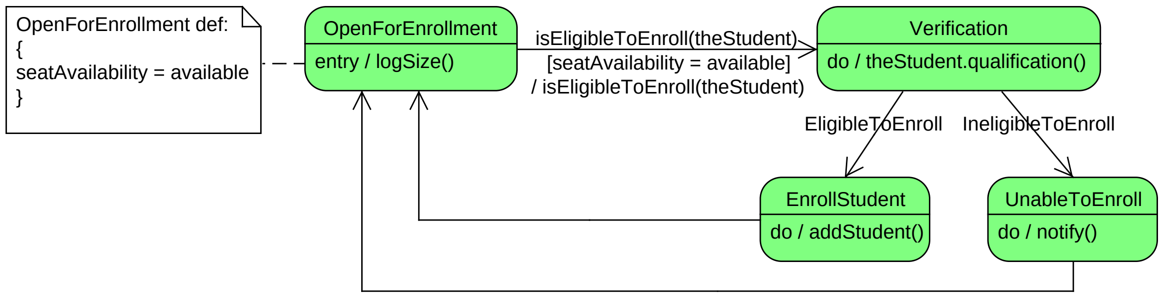


Figure 5.25: Augmented CGs model after applying call event inference rule

Thus the generated incomplete model has fulfilled its purpose of inviting the modeler to provide additional requirements knowledge which will become a part of the collection of models for a given system being developed.

5.2.3 Generating Sequence Diagrams from the CGs Reservoir

In this subsection, the CGs Reservoir is used to generate sequence diagrams. We assume that the CGs Reservoir includes the requirements knowledge of class and state diagrams expressed in CGs already.

5.2.3.1 Inference Rules for Sequence Diagrams

Since sequence diagrams describe messages passing among objects, the key to generating sequence diagrams is to look for messages and their temporal sequences in the CGs Reservoir. Luckily, both *Message* and *Time* are primitives, so they do not need to be inferred from the CGs Reservoir. However, inference rules for sequence diagrams are still needed to make the generation easier. In Figure 5.26, two inference rules are developed to infer sender or receiver when a message is found in the CGs Reservoir.

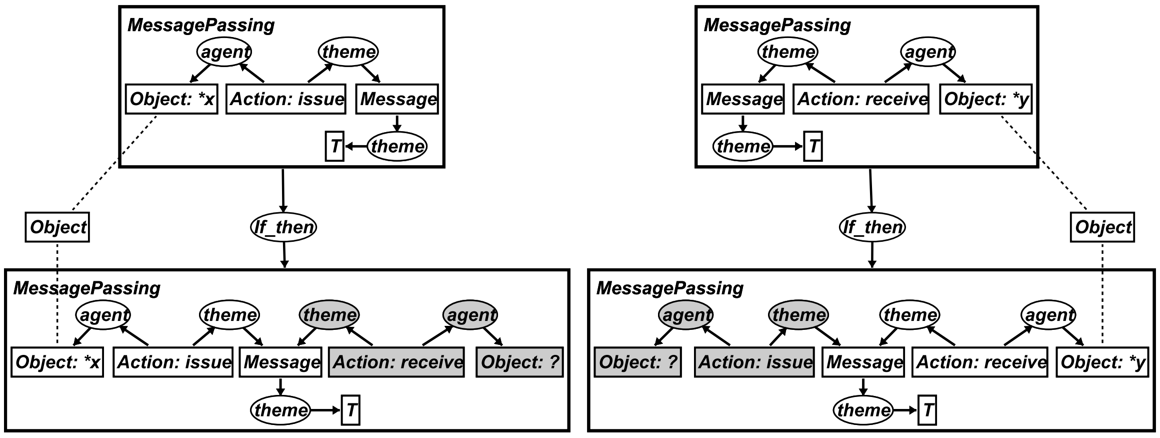


Figure 5.26: Sequence diagram inference rule set

5.2.3.2 Inferring Sequence Diagrams from CGs

We will now demonstrate how to use sequence diagram inference rules to generate sequence diagrams from the CGs Reservoir. The CGs Reservoir contains requirements knowledge of the class diagram and the state diagram of the UnivSys. case study.

The inference rules in Figure 5.26 are used to make sure that every message found in the CGs Reservoir has both a sender and a receiver. Once this is done, we can extract all sequences (temporal sequences) from the current CGs Reservoir. Because of the size of this model, the generated sequence CGs model is shown in two figures (Figure 5.27 and Figure 5.28).

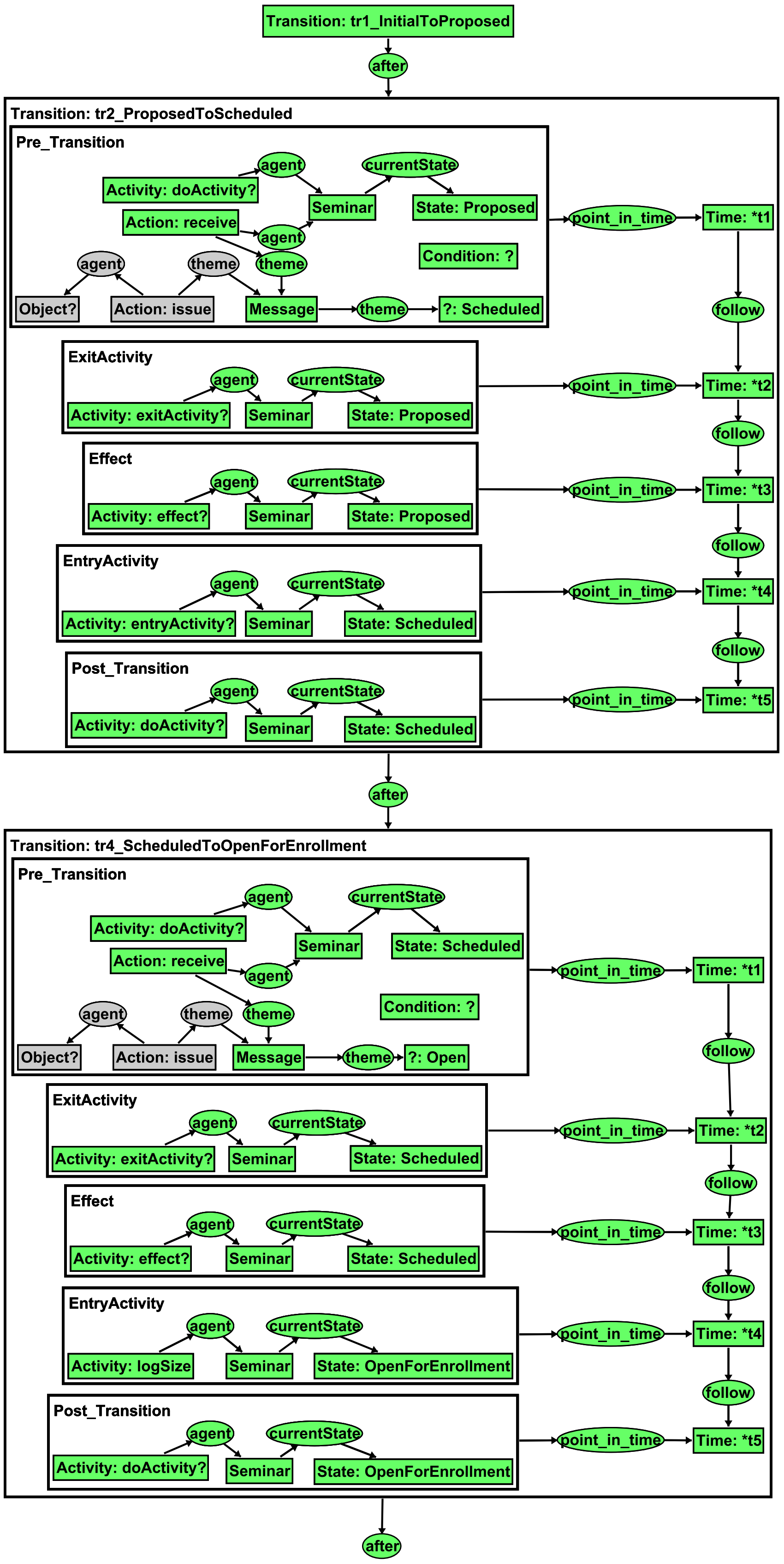


Figure 5.27: Generated CGs sequence model part 1

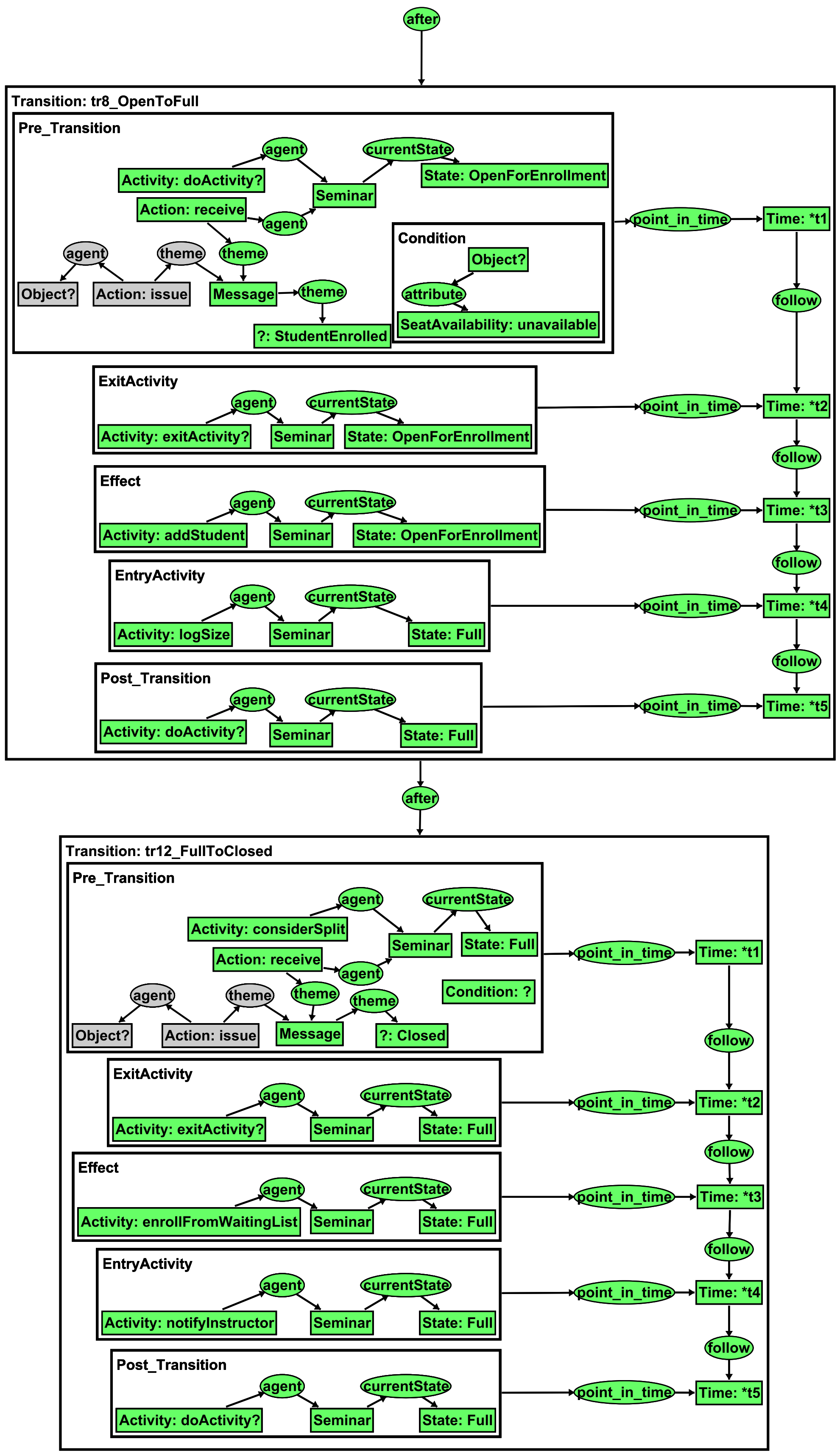


Figure 5.28: Generated CGs sequence model part 2

New requirements knowledge (shown as gray concepts and relations) is added to the existing CGs in the CGs Reservoir (shown as green concepts and relations). We want to remind the readers that green represents the requirements knowledge of state diagrams in the CGs Reservoir.

The generated CGs in Figure 5.27 and Figure 5.28 are then translated back to UML sequence diagram notations (Figure 5.29). Several semantic holes are identified: unknown lifelines sending messages to `Seminar` need to be filled in and multiple execution specifications along the lifeline of `Seminar` need to be renamed.

This generated incomplete diagram clearly shows a knowledge acquisition opportunity: a sequence diagram modeler presented with the generated sequence diagram in Figure 5.29 can fill in the semantic holes by providing additional information. A possible sequence diagram that might result from resolving semantic holes by a sequence diagram modeler is shown in Figure 5.30. Note that the three different lifelines sending `Scheduled`, `Open`, and `Closed` messages are merged into one lifeline `SeminarEnrollment` whose role is as a controller. New messages like `create` and `considerSplit` are added. A combined fragment `alt` frame with two guards is added to make the scenario more reasonable. Several `entry`, `exit`, `do` activities and `effects` are either deleted after consideration or renamed.

Thus the generated incomplete model has fulfilled its purpose of inviting the modeler to provide additional requirements knowledge which will become a part of the collection of models for a given system being developed.

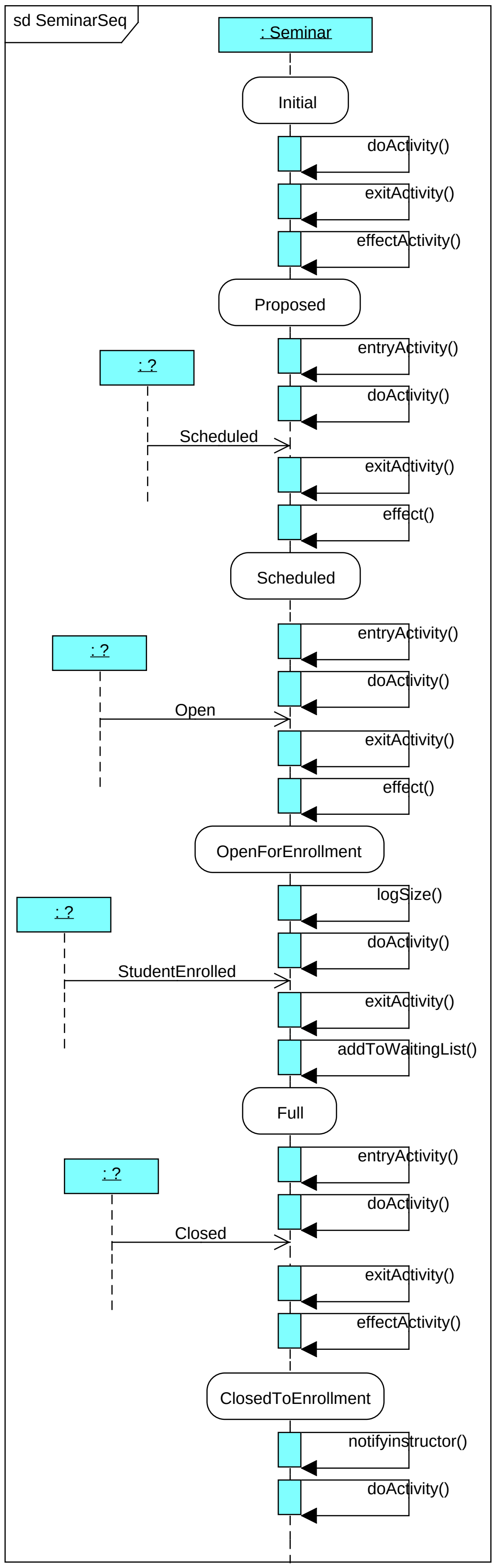


Figure 5.29: Generated sequence diagram with semantic holes

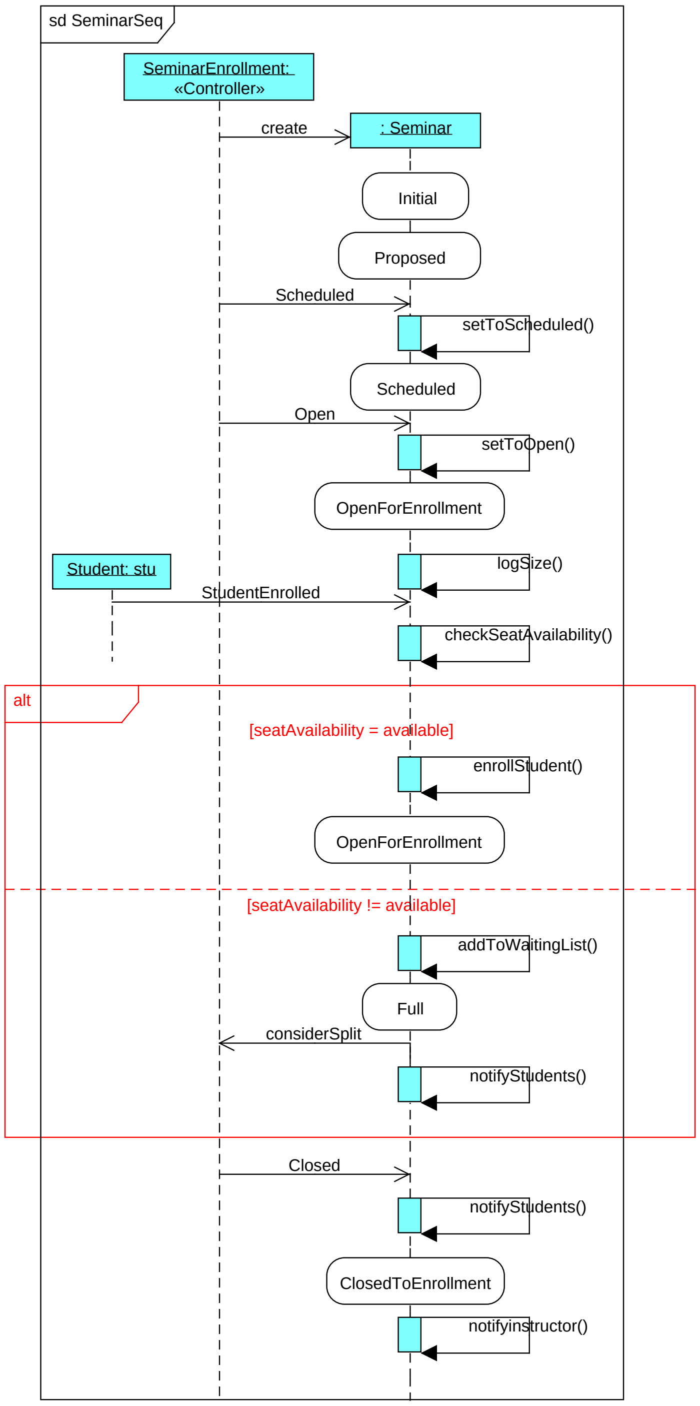


Figure 5.30: Sequence diagram after completion

5.3 Summary

By now, we have finished the development of CRF using CGs as a common representation of UML diagrams of a system. With CRF, UML diagrams in the requirements “ecosystem” can be converted to CGs according to canonical graphs and the CGs can be translated back to UML diagrams according to inference rules. During the transformation, semantic holes are identified and used as knowledge acquisition opportunities to acquire more requirements from UML diagram modelers. At this point, readers are able to look at the CRF portion of the three complete case studies in the appendices, they are Section A.2, Section B.2 and Section C.2 in the appendices. As readers can see, the CRF portion of each case study is much longer than the PWF portion.

CHAPTER 6

A COMPARISON OF THE TWO FRAMEWORKS

In this work, two frameworks that can facilitate requirements acquisition during multiple-viewed requirements modeling have been proposed and carefully described in previous chapters. By carrying out different kinds of transformations, both frameworks can present requirements modelers with generated analysis models that have semantic holes so that they can start to fill in those holes or modify them. Since the two frameworks have well-defined model transformation processes and are based on the same semantics of UML diagrams, a comparison is possible. As already mentioned in Chapter 3, we have applied the two requirements acquisition frameworks to three non-trivial case studies. The three case studies have gone through the transformation processes as described in Chapter 4 and Chapter 5 manually. The results of the three case studies are quite lengthy and therefore are shown in the appendices. The comparison of the two frameworks in this chapter is based on the those results.

In this chapter, we propose a set of framework-independent criteria (Table 6.1) that can be used to evaluate the effectiveness in transforming and acquiring requirements knowledge in the two frameworks; we then compare them based on the evaluation results. Note that Table 3.1 in Chapter 3 is reprinted here for convenience.

These criteria are not limited to the two frameworks developed in this dissertation; they are meant to be applied to any framework that claims to address the requirements knowledge acquisition problem in the context of multiple-viewed requirements modeling. The criteria are divided into two categories: quantitative and qualitative. Section 6.1 to Section 6.6 present the comparison results of each criterion based on the results of the three case studies in the appendices.

Table 6.1: Criteria for evaluating requirements acquisition frameworks

NO.	**Criterion name**	**Criterion description**
	Quantitative criteria	
1	Capability of acquiring missing requirements	This criterion measures the number of the missing requirements that can be potentially acquired from a modeler given a generated UML diagram.
2	Capability of generating definite requirements	This criterion measures the number of the definite requirements generated in a generated UML diagram from other existing UML diagrams.
3	Percentage of the missing requirements in generated UML diagrams	This criterion represents the percentage of the missing requirements in a generated UML diagram.
4	Extensibility	This criterion evaluates the ability to include a new type of UML diagram in the framework.
5	Knowledge acquisition effort	This criterion measures the effort of eliciting knowledge from requirements modelers.
	Qualitative criteria	
6	Capability of reasoning in requirements knowledge	This criterion determines whether or not the framework provides reasoning capability or not.

6.1 Capability of Acquiring Missing Requirements

Both frameworks can generate UML diagrams with semantic holes. Since the presence of semantic holes indicates the possible missing requirements knowledge, a high number of semantic holes in a UML diagram is a sign that more requirements knowledge will be potentially acquired from a modeler. This criterion evaluates the capability that a framework has to acquire the missing requirements. This is measured by counting the number of semantic holes in the UML diagrams generated by a framework. An advantage of counting semantic holes is that this only depends on the representation of UML diagrams and not on any subjective judgment of incompleteness or experience of requirements modelers, since these requirements (semantic holes) are missing for sure, and need to be provided by the requirements modeler.

During the comparison, given the same state and sequence diagrams of a software system, two class diagrams are generated by PWF and CRF, respectively; then the number of semantic holes yielded in each of the two generated class diagrams is counted (see the five rows under the gray row "In generated class diagram" in Table 6.2). The same comparison process works for generating state and sequence diagrams (see the rest of Table 6.2). The complete results of evaluating the generated class, state and sequence diagrams for three case studies in two frameworks are listed in Table 6.2.

Table 6.2: Comparison of the capability of acquiring missing requirements in two frameworks

	PWF			**CRF**		
Semantic holes	UnivSys.	Cryptanlys.	MineSys.	UnivSys.	Cryptanlys.	MineSys.
In generated class diagram						
Number of unknown class names	7	8	4	7	8	6
Number of unknown attribute names	0	0	0	15	20	25
Number of unknown operation names	13	14	27	49	59	81
Number of unknown association names	12	11	13	12	11	15
Total	32	33	44	83	98	127
In generated state diagram						
Number of unknown/potential states	20	18	19	20	18	13
Number of unknown transitions	4	3	5	5	4	3
Number of unknown events	5	3	2	5	4	3
Number of unknown effects	0	0	0	7	7	5
Number of unknown guards	0	0	0	15	14	10
Number of unknown entry/exit, do activities	0	0	0	60	54	39
Number of state invariants	0	0	0	10	10	7
Total	29	24	26	122	112	80
In generated sequence diagram						
Number of unknown neighboring lifelines	6	8	8	6	7	8
Number of unknown messages	0	1	0	0	0	0
Number of unknown execution specs	0	0	0	25	37	58
Total	6	9	8	31	44	66

6.1.1 Quality of the Semantic Holes Generated by CRF

During the comparison of criterion 1, we found that in CRF, semantic holes are generated in two ways: by instantiating canonical graphs and by knowledge inference (Table 6.3).

Table 6.3: Types of semantic holes in generated UML diagrams by CRF

Semantic holes category	**Source**
Potentially useful semantic holes	By canonical graphs (template)
Obviously useful semantic holes	By knowledge inference

In CRF, the canonical graphs are used to translate UML diagrams to CGs; readers may have already realized that canonical graphs are actually templates that represent the expected concepts and relations of a UML diagram when it is translated in CGs form. So the presence of unfilled slots in canonical graphs indicates the incompleteness of a UML diagram. For example, by instantiating the canonical graph of a state transition (Figure 6.1), the states involved in a transition in the state diagram are asserted to own `do activity`, `entry/exit` activity, and the state transition is asserted to own `event`, `guard`, `effect`, even if those model elements were originally not in the state diagram. For example, Figure 6.2. This is not to say that they must be specified in this UML diagram; the value of potentially useful holes is that the modelers can be made aware of the model elements that are necessary to construct a UML diagram [6]. Semantic holes generated by instantiating canonical graphs are called potentially useful semantic holes, since they probably turn out to be empty and deleted by the modeler.

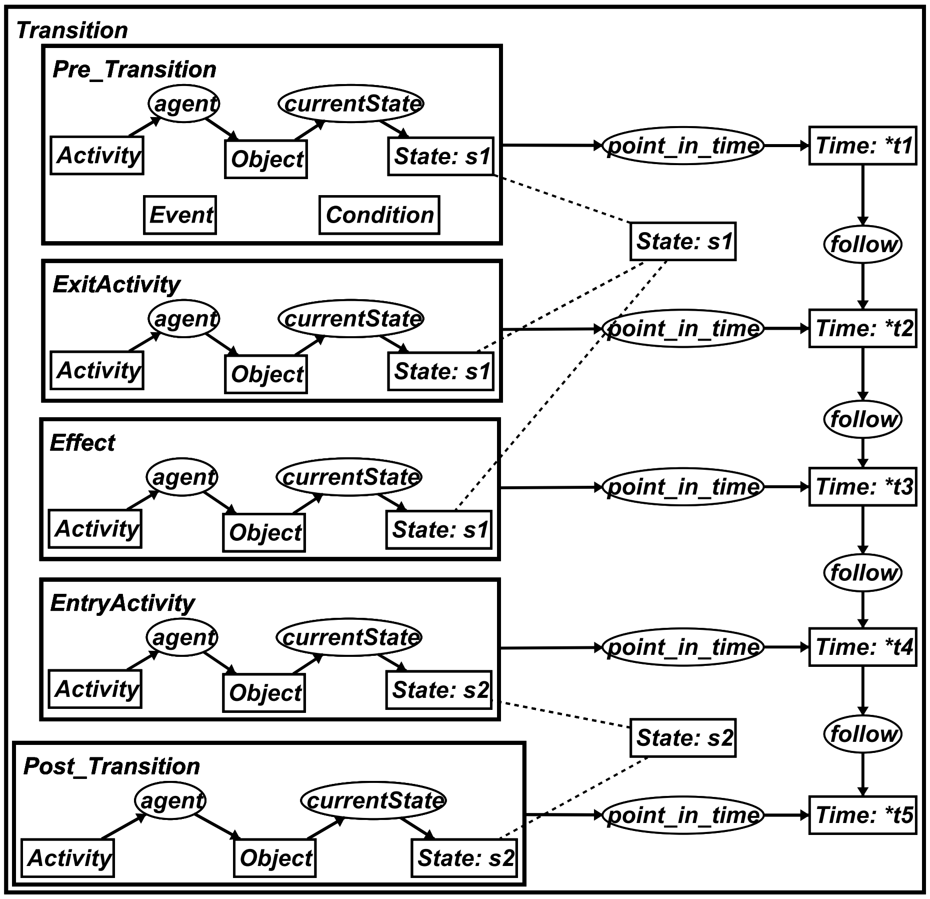


Figure 6.1: Template nature of the canonical graph of UML diagram

The other kind of semantic holes is generated by applying inference rules defined in the CGs Support. For example, the inference rule in Figure 6.3 will assert the existence of a sender concept given the fact that a *Message* type concept is received by a *Object* type concept. Such semantic holes are called obviously useful semantic holes in this work since they are guaranteed to exist and must be filled explicitly by a modeler. An obviously useful semantic hole is more valuable than a potentially

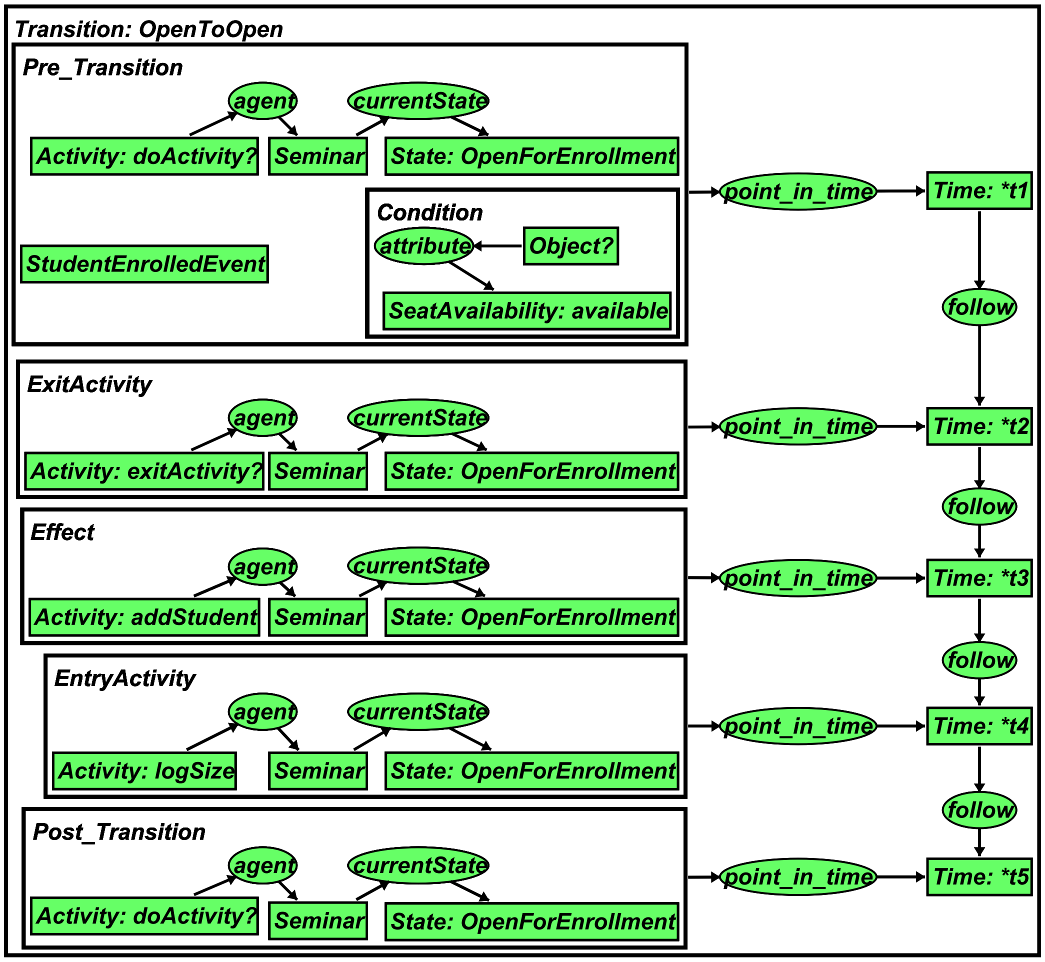


Figure 6.2: Potentially useful semantic holes

useful semantic hole. The percentage of the obviously useful semantic holes and the potentially useful semantic holes in generated UML diagrams is shown in Table 6.4.

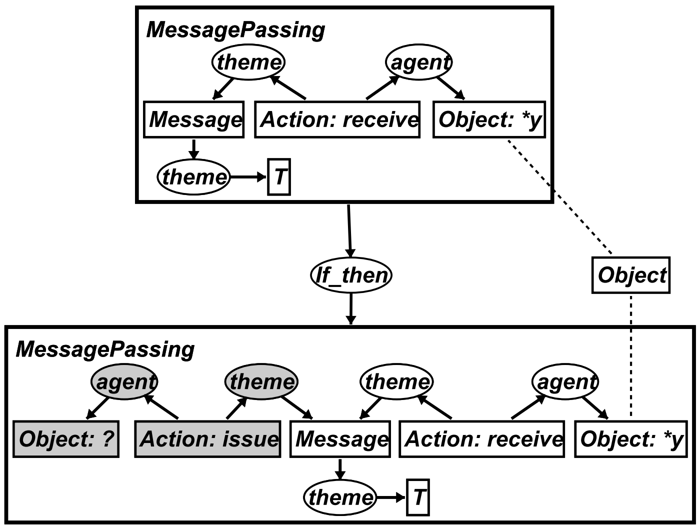


Figure 6.3: Obviously useful semantic holes

Table 6.4: Percentage of two kinds of semantic holes in generated UML diagrams by CRF

Generated UML diagrams	**Potentially useful semantic holes**			**Obviously useful semantic holes**		
	UnivSys.	Cryptanlys.	MineSys.	UnivSys.	Cryptanlys.	MineSys.
Class diagram	55.4%	55.1%	57.5%	44.6%	44.9%	42.5%
State diagram	71.3%	71.4%	71.2%	28.7%	28.6%	28.8%
Sequence diagram	80.6%	84.1%	87.9%	19.4%	15.9%	12.1%

As we can see from Table 6.4, potentially useful semantic holes account for a large portion of the overall semantic holes generated by CRF. Since PWF in this work does not generate potentially useful semantic holes, for the sake of fairness, we conducted the comparison again after subtracting potential useful semantic holes (Table 6.5).

Table 6.5: Comparison of the capability of acquiring missing requirements in two frameworks without considering potentially useful semantic holes

Generated UML diagrams	**PWF**			**CRF**		
	UnivSys.	Cryptanlys.	MineSys.	UnivSys.	Cryptanlys.	MineSys.
Class diagram	32	33	44	37	44	54
State diagram	29	24	26	35	32	23
Sequence diagram	6	9	8	6	7	8

6.2 Capability of Generating Definite Requirements

In a generated UML diagram, besides semantic holes, there are newly generated requirements that we are sure of based on inference rules. These requirements are called definite inferred requirements and do not need further clarification by modelers. This criterion evaluates the capability that a frameworks has to acquire the definite requirements knowledge. This is measured by counting the number of definite model elements in the UML diagrams generated by a framework. Similar to criterion

1, during the comparison, UML diagrams are generated by PWF and CRF, respectively, but then we count the number of new definite model elements yielded in the generated UML diagrams. The results of evaluating the generated class, state and sequence diagrams for three case studies in two frameworks are listed in Table 6.6.

Table 6.6: Comparison of the capability of generating definite requirements in two frameworks

	PWF			**CRF**		
Semantic holes	UnivSys.	Cryptanlys.	MineSys.	UnivSys.	Cryptanlys.	MineSys.
In generated class diagram						
Number of definite classes acquired	5	4	8	5	4	8
Number of definite attributes acquired	2	3	11	1	2	5
Number of definite operations acquired	11	9	0	10	9	0
Number of definite association names acquired	0	0	0	0	0	0
Total	18	16	19	16	15	13
In generated state diagram						
Number of definite state machines	5	4	5	5	4	5
Number of definite states	0	0	0	0	0	0
Number of definite transitions	15	14	9	10	10	7
Number of definite events	10	8	7	10	10	7
Number of definite effects	9	9	7	9	9	5
Total	39	35	28	34	33	22
In generated sequence diagram						
Number of definite sequences	4	2	4	9	9	11
Number of definite messages	6	6	6	6	6	6
Number of definite execution specs	5	4	0	5	3	0
Number of definite states	6	10	14	6	10	14
Number of definite combined fragments	2	2	10	0	0	0
Total	23	24	34	26	28	31

6.3 Percentage of the Missing Requirements in Generated UML Diagrams

This criterion evaluates the incompleteness of the UML diagrams generated by PWF and CRF. The incompleteness of a generated UML diagram is the number of semantic holes over the number of overall generated model elements in a generated UML diagram. The results of evaluating the incompleteness of generated class, state, and sequence diagrams for three case studies in two frameworks are listed in Table 6.7. For example, for the generated class diagram of the UnivSys. by CRF, it is 83.84% incomplete and 16.16% complete.

Table 6.7: Comparison of incompleteness of generated UML diagrams by two frameworks

Generated UML diagrams	**PWF**			**CRF**		
	UnivSys.	Cryptanlys.	MineSys.	UnivSys.	Cryptanlys.	MineSys.
Class diagram	64.00%	42.65%	20.69%	83.84%	78.21%	54.39%
State diagram	67.35%	40.68%	27.27%	86.73%	77.24%	61.11%
Sequence diagram	69.84%	48.15%	19.05%	90.71%	78.43%	68.04%

6.4 Extensibility

In this work, we choose three types of UML diagrams for both frameworks. However, more UML diagrams can be added to expand the frameworks. This criterion evaluates the extensibility of a framework by measuring the amount of effort needed to introduce another type of UML diagram (activity diagram) in both frameworks. For simplicity, only limited model elements in activity diagrams are considered:

- Activity partitions;
- Activity nodes and control flows;
- Forking and joining.

6.4.1 Adding Activity Diagrams to PWF

This subsection describes the process of accommodating activity diagrams in PWF.

6.4.1.1 From Activity Diagrams to State Diagrams

A UML activity diagram specifies an activity in which multiple objects are involved and data would be created or modified. Each flow transition may affect several objects' state machines. Multiple state machines then will be generated from one activity diagram. Detailed transformation rules are summarized in Table 6.8.

6.4.1.2 From State Diagrams to Activity Diagrams

In a state diagram, only one object is considered; this object might stay in a state waiting for triggering events without performing any activity or it might perform an ongoing activity until completion or being interrupted when triggering events

Table 6.8: Rules for transforming activity diagrams to state diagrams

Model element(s) in activity diagrams	**Corresponding element(s) in state diagrams**
Activity partitions	Different state machines
Activity nodes	States with a do activity
Send nodes	Send actions
Receive nodes	Triggers for state transitions
Control flows	Completion transitions

occur. As a result, knowledge of activity nodes and their ordering can be derived from state diagram. Detailed transformation rules are summarized in Table 6.9.

Table 6.9: Rules for transforming state diagrams to activity diagrams

Model element(s) in state diagrams	**Corresponding element(s) in activity diagrams**
State machines	Activity partitions
Signal events	Receive nodes
Entry/exit, do activities and effects	Activity nodes
State transition paths	Control flow paths

6.4.1.3 From Activity Diagrams to Class Diagrams

In an activity, multiple objects are involved and some data objects would be created or modified. Such information should be described in a class diagram. Detailed transformation rules are summarized in Table 6.10.

Table 6.10: Rules for transforming activity diagrams to class diagrams

Model element(s) in activity diagrams	Corresponding element(s) in class diagrams
Activity partition	Class definitions
Activity nodes	Operations of the class
Control flows	A flow edge across activity partitions indicates association between classes
Control nodes	Variables in control nodes are treated as attributes of the class
Object flows	Objects produced and consumed in the activity diagram are treated as classes

6.4.1.4 From Class Diagrams to Activity Diagrams

In a class diagram, objects with the same attributes and behaviors are described by a class. The relationships between different objects are described by the relationships between their corresponding classes. Figuring out the orders of actions is an important part of deriving activity diagrams from class diagrams. However, similar to transforming class diagrams to state diagram or sequence diagram in Chapter 4, it is impossible to infer the control flow knowledge needed among actions given just the structural information. As a result, we could not develop rules to transform class diagrams to activity diagrams.

6.4.1.5 From Activity Diagrams to Sequence Diagrams

Activity diagrams and sequence diagrams have many things in common. Detailed transformation rules are summarized in Table 6.11.

Table 6.11: Rules for transforming activity diagrams to sequence diagrams

Model element(s) in activity diagrams	Corresponding element(s) in sequence diagrams
Activity partitions	Lifelines
Activity nodes	Execution specifications
Send nodes	Asynchronous outgoing messages
Receive nodes	Asynchronous incoming messages
Control flows	Sequence of messages

6.4.1.6 From Sequence Diagrams to Activity Diagrams

Detailed transformation rules are summarized in Table 6.12.

Table 6.12: Rules for transforming sequence diagrams to activity diagrams

Model element(s) in sequence diagrams	Corresponding element(s) in activity diagrams
Lifelines	Activity partitions
Synchronous messages	Activity nodes
Asynchronous messages	Send nodes
Synchronous message arguments	Object flows
Asynchronous message arguments	Object flows
Return message with value	Object flows
Sequence of messages	Control flows

6.4.2 Adding Activity Diagrams to CRF

When a new type of UML diagram is introduced in CRF, the CGs Support needs to be extended to support the conversion and generation of the new UML diagram. In particular, new primitives are added to facilitate the conversion process and new inference rules are developed to support the generating process. By studying the semantics of activity diagrams [32], we try to represent each model element of activity diagrams using the current primitives in the CGs Support. If a model element cannot be represented, then new primitives need to be added.

Activity nodes are the basic elements of activity diagrams. We can use the primitive type *Activity* to describe activity nodes in activity diagrams. The activity nodes within an activity may be organized into partitions, often called swimlanes. In CRF, we use primitive type *Object* to represent activity partitions. Control flow that connects activity nodes is represented by primitive type *Time* and primitive relations *point_in_time* and *follow* in CRF. CGs in Figure 6.4 represents two activity nodes connected by a flow and organized in two different partitions.

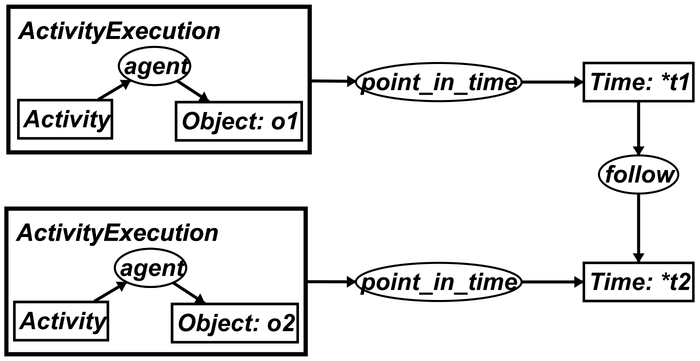


Figure 6.4: Canonical graph for activity nodes and control flows

Besides regular activity nodes, there are special activity nodes in activity diagrams: send and receive signals (see Figure 6.5).

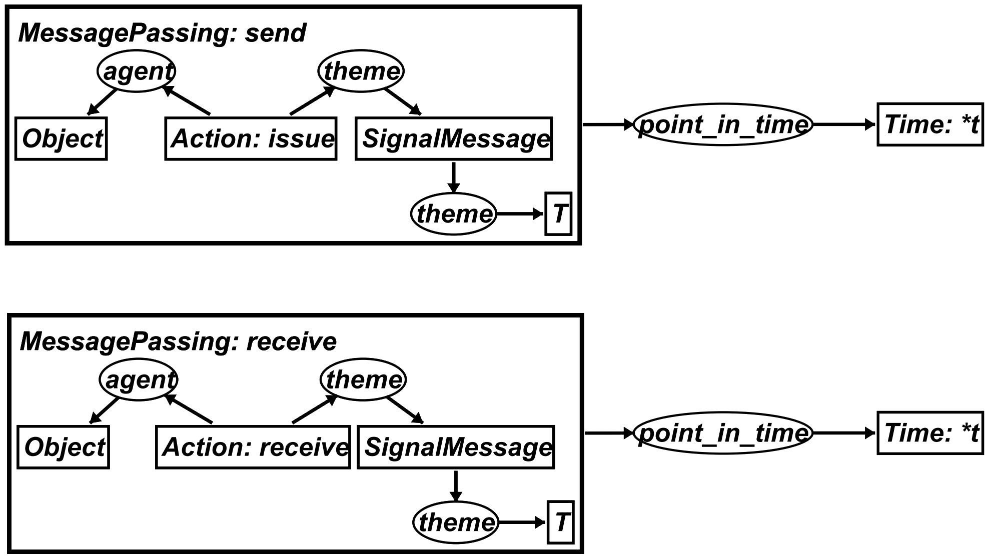


Figure 6.5: Canonical graph for sending and receiving signals

Concurrency is an important feature in activity diagrams and is represented by forking and joining in Figure 6.6.

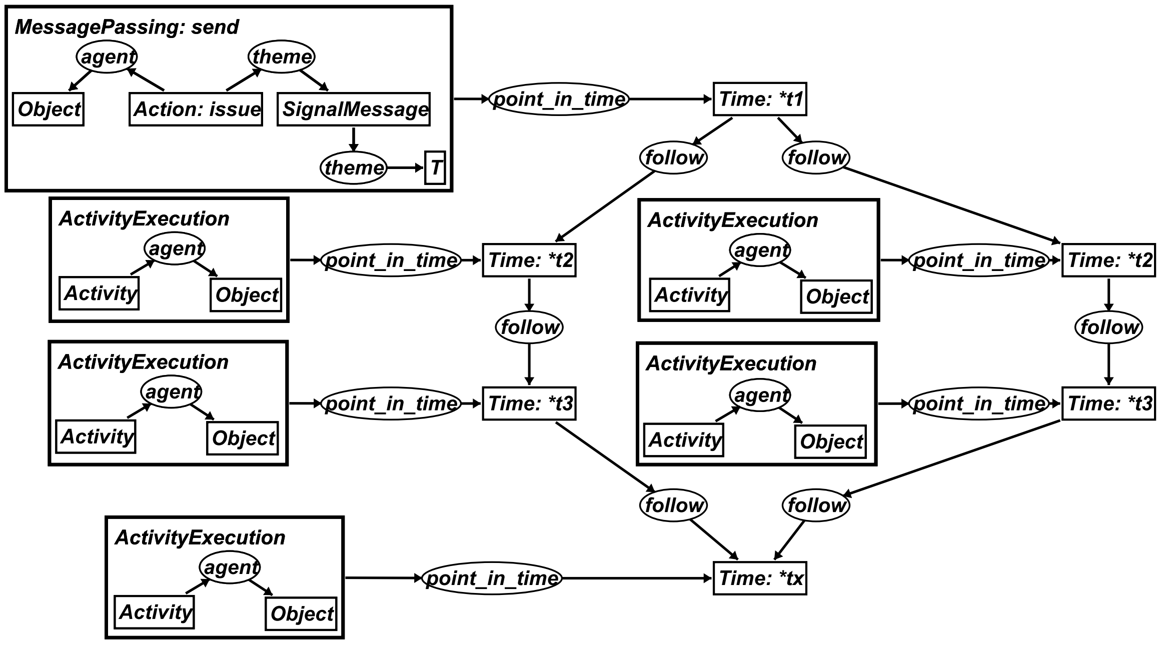


Figure 6.6: Canonical graph for forking and joining

Inference rules for generating an activity diagram's CGs model are developed by considering the semantics of an activity diagram. Particularly, we look for the primitive concept *Activity* and *Action* in the CGs Reservoir. The objects that perform them will become different activity partitions in the activity diagram.

6.4.3 Comparison of Extensibility

In summary, the effort involved in extending a framework to accommodate a fourth UML diagram is compared in Table 6.13.

Table 6.13: Comparison of extensibility of the two frameworks

Development effort	**PWF**	**CRF**
New rules developed	26	4

6.5 Knowledge Acquisition Effort

This criterion is used to evaluate and compare the amount of effort needed to complete semantic holes in the UML diagrams generated by PWF and CRF. When a modeler is presented with UML diagrams with semantic holes, he/she needs to look at each hole and choose to either fill it in or delete it. So, in this work, the knowledge acquisition effort is measured by counting the number of semantic holes that needed to be considered (Table 6.14).

Table 6.14: Comparison of the knowledge acquisition effort needed in UML diagrams generated by two frameworks

Generated UML diagrams	**PWF** UnivSys.	Cryptanlys.	MineSys.	**CRF** UnivSys.	Cryptanlys.	MineSys.
Class diagram	32	33	44	83	98	127
State diagram	29	24	26	122	112	80
Sequence diagram	6	9	8	31	44	66

6.6 Capability of Reasoning in Requirements Knowledge

During the model transformation process, the CRF results in a central CG Reservoir which contains the requirements knowledge of UML diagrams. This knowledge reservoir can be used for other purposes, such as inconsistency checking, requirements integration, and inquiry. The PWF does not have this advantage.

6.7 Summary

In this chapter, we conducted a comparison of the PWF and CRF according to the six criteria we proposed. Criterion 1 compared the capability of acquiring missing requirements of the two frameworks; based on the results of applying both frameworks to three case studies, the CRF outnumbered the PWF in all generated UML diagrams in three case studies (see Table 6.2). In other words, using CRF, more possible requirements may be acquired from a modeler in the multiple-viewed requirements modeling context. Readers may argue that a portion of the semantic holes generated in CRF (see Table 6.4) is due to the template nature of canonical graphs (such holes are called potentially useful semantic holes), so, by instantiating a canonical graph, many "slots" are inferred. For fairness, we compared the two frameworks again in Table 6.5 without considering the potentially useful holes generated in CRF and found that the CRF still outnumbered the PWF. However, we firmly believe that the potential useful semantic holes are useful for the purpose of knowledge acquisition. We have no way to measure the usefulness of a potential useful semantic hole as this is out of scope in the current work.

Criterion 2 compared the capability of generating definite requirements in the two frameworks; based on the results of applying both frameworks to three case studies, PWF outnumbered CRF in all generated UML diagrams in three case studies. In other words, using PWF, more determined requirements can be generated. This implies that PWF is better at generating target models from source models.

Built upon the previous two criteria, the comparison based on criterion 3 shows that UML diagrams generated in CRF are generally more incomplete than in PWF. This matches our first conclusion that CRF is more capable of exposing knowledge acquisition opportunities in the multiple-viewed requirements modeling context.

Criterion 4 is crucial for framework developers since a framework grows when new types of diagrams are introduced; based on our experiment of accommodating activity diagrams in both frameworks, CRF is clearly more extensible than PWF. This advantage becomes more evident when a framework supports more UML diagrams.

More knowledge acquisition opportunities mean more effort involved in resolving them. In CRF, a modeler has to go through each semantic hole, particularly potential useful holes to decide if it is indeed a semantic hole. We conclude that it takes more time to complete a UML diagram generated by CRF than by PWF because more holes need to be taken into account.

Besides requirements acquisitions, CRF has the advantage of reasoning on the knowledge base. Criterion 6 indicates that more reasoning can be performed on the CGs Reservoir. However, PWF does not have any knowledge repository; it is purely used for model transformations.

CHAPTER 7

DISCUSSION

In this chapter, we discuss the strengths and limitations of this work, explain some questions that readers may have, and point to future work.

7.1 Two well-founded frameworks for requirements acquisition

In this work, two frameworks (PWF and CRF) are developed to generate UML diagrams with semantic holes for the purpose of requirements knowledge acquisition. They are well-founded and based on the standard UML semantics. Our evaluation and comparison have shown their effectiveness in requirements acquisition in multiple-viewed requirements modeling.

The PWF depends on pair-wise transformation rules and is quite straightforward. The design of CRF in this work, however, needs some discussion here. The CGs Reservoir in CRF can be considered heterogeneous in that it is able to store requirements knowledge from different types of UML diagrams, each of which conveys quite different requirements; it also can be considered a homogeneous knowledge reservoir since all requirements knowledge is represented using the same set of primitive concepts and relations which is fundamental in an object-oriented requirements specification. In other words, for each requirement in the UML diagrams we used in this work, its semantics can be represented by a CG that uses only the concepts and relations in the Primitives of the CGs Support. This feature is crucial in our work, since the CGs Reservoir only needs to deal with CGs constructed from the primitive concepts and relations. The usage of primitives as the foundation of converting from UML diagrams to CGs greatly reduces the complexity of the resulting CGs Reservoir and achieves a fine-grained and cohesive central requirements knowledge reservoir. This is a significant improvement and simplification of previous work, especially in [2]. In this work, the primitives we developed are not tied to a specific kind of UML diagram and they are well justified from previous work on underlying elements in software requirements [25] [26].

7.2 Limitations of the CRF

The current primitive concepts and relations work well for the three kinds of UML diagrams. We do not claim the completeness of this set of primitives, since we have not considered all UML diagrams, and, furthermore, the current primitives and canonical graphs are not sufficient to convert some complex model elements used in the three UML diagrams we have considered. For example, we do not yet support converting association classes in class diagrams, combined fragments like `loop` and `alt` in sequence diagrams, or nested states, concurrent states, and history states in state diagrams. New primitive concepts and relations must be added in future to accommodate those features.

The current work assumes its input UML diagrams are syntactically and semantically correct. In other words, they are well-formed according to UML syntactic constraints. If errors exist, those errors will be converted to CGs and then used to infer incorrect or badly formed requirements knowledge. It is possible that automatic inference techniques in CGs would be able to detect these, but that is clearly beyond the scope of the current work. Also, retracting the CGs inferred from this wrong knowledge is a time-consuming and complex task that the current framework cannot handle. We consider this a minor limitation, since existing UML tools are quite capable of ensuring valid UML diagrams are produced before we ever attempt our translation into CGs.

Another current limitation is the lack of automation support. Of course, manually converting UML diagrams of a system under development to CGs and manually generating new UML diagram from CGs Reservoir would both be tedious and error-prone processes. Future work obviously will focus on automating these.

7.3 Two Types of Semantic Holes in CRF

For criterion 1 (Capability of acquiring missing requirements), CRF greatly outnumbered PWF in all UML diagrams in three case studies (see Table 6.2) because a large portion of semantic holes are generated by asserting canonical graphs (see Section 6.1.1). They are named potential usefully semantic holes in that many of them may turn out to be empty and discarded. For example, not all state transitions have guards and effects. However, we still believe that these potentially useful semantic holes are "useful" for the purpose of knowledge acquisition. The real value of potentially useful holes is the thinking behind them. A potentially useful semantic hole may not be a real missing requirement in a target UML diagram, but it might provide enough insight to elicit requirements. In the current work, we have no way to measure the usefulness of a potentially useful semantic hole. One promising approach is that we can introduce weighted semantic holes in the CRF; the weight represents the quality of the semantic hole. In other words, the higher the weight is, the more likely the semantic hole is useful. The weight can be obtained by querying in a large requirements library with similar projects.

7.4 Supporting Other UML Diagrams in Two Frameworks

More UML diagrams can be added in our knowledge acquisition frameworks. We already outlined an extensibility approach in Section 6.4. We intend to continue that approach to introduce more UML diagrams in both frameworks such as object diagrams, communication diagrams, and use case diagrams. For PWF, several previous work has already figured out the pair-wise transformation rules, but, for CRF, we need to figure out how to translate those UML diagrams to and from CGs. Here is our initial approach: object diagrams consist of objects and links that can be represented as *Object* type concepts and *association* type relations. A communication diagram is another kind of interaction diagram and has very similar semantics as a sequence diagram, its CGs are similar to that of sequence diagram. Both object and communication diagrams can be accommodated in CRF without introducing additional primitive concepts and relations in the CGs Support. Use case diagrams depict the interactions between the system and users. More primitive concept types like

Use case and *Actor* and relation types like *include* and *extend* need to be added to expand the current CGs Support.

7.5 The Size of CGs Expressing Semantics of UML Diagrams

We are sure that readers are impressed by the sheer size of the CGs in this work (look at the number of pages in appendices). The whole point of having the Primitives in the CGs Support is that a diagram can be expressed in CGs using low-level concepts and relations defined in it. The primitives are to modeling languages (e.g. UML) as assembly language statements are to high-level programming languages. As a result, a semantically rich UML diagram takes more time and space to be expressed in primitives because each model element contains a lot of semantics. For example, a transition in state diagrams is a simple line with arrow but it turns out to take nearly one page when its semantics are expressed in CGs. On the other hand, if model elements in a diagram are close to primitives, the size of its corresponding CGs would be small.

7.6 Working with a Requirements Acquisition Methodology

We introduced the idea of conceptual feedback in Chapter 1. Two frameworks, PWF and CRF, can be used to implement this idea (Figure 7.1 shows the CRF).

During conceptual feedback, diagrams in the requirements "ecosystem" have different degrees of completeness and we can choose one UML diagram to start with and make it more complete by augmenting it with requirements from all other UML diagrams in the "ecosystem." By filling in the holes and adding additional require-

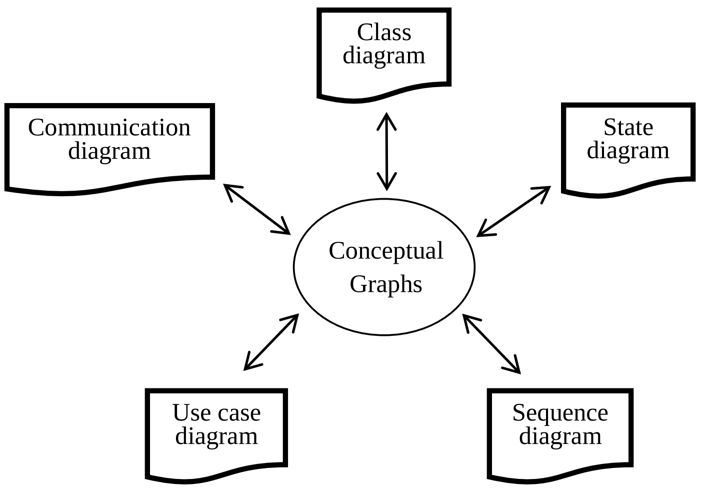


Figure 7.1: Requirements acquisition methodology in CRF

ments knowledge in this diagram, it becomes more complete than it was. A second UML diagram in the requirements "ecosystem" then is picked and the same process is repeated. Sooner or later, all diagrams become more and more complete, and the growth of the knowledge of the entire requirements in the "ecosystem" becomes slow because there are not many holes through transformations. A natural question a reader might ask here is, what is the best order of picking and transforming? Figuring out what is the best order to make this work well is an interesting topic but it is not our focus. A future work would consider using the CRF in a working environment and figure out how the order of development of the diagrams can affect the effectiveness in requirements knowledge acquisition in the requirements "ecosystem."

7.7 Future work

Future work involves resolving automating the requirements acquisition process in PWF and CRF, including more UML diagrams in both frameworks and figuring our an effective acquisition order for several UML diagrams in the requirements "ecosystem."

CHAPTER 8

CONCLUSION

Determining what is missing in requirements analysis models is normally a difficult task, but this is made easier by using model transformations during multiple-viewed requirements modeling. In this work, we have described a pair-wise framework and developed a CGs-based common representation framework in which UML diagrams with semantic holes are generated for the purpose of requirements acquisition. The presence of semantic holes in the generated UML diagrams makes modelers aware of the need to look for specific things and provides an excellent opportunity for further elicitation of new requirements knowledge. The two knowledge acquisition frameworks have been successfully used in three case studies and, based on the results, a comparison according to 6 criteria are conducted.

In conclusion, this work contributes to the software engineering community and the knowledge engineering community in two ways:

1. We proposed and developed the CGs-based CRF for requirements acquisition during multiple-viewed modeling to demonstrate that knowledge-based model transformations can indeed provide effective assistance to the critical requirements process by acquiring more requirements knowledge.

2. We provide a set of useful criteria for evaluating frameworks that can address the requirements acquisition problem in the context of multiple-viewed modeling, so that requirements modelers can make decisions and trade-offs based on those criteria when choosing knowledge acquisition frameworks for acquiring requirements knowledge. Moreover, the criteria in this work serve as a baseline for future work and more criteria can be included.

APPENDICES

APPENDIX A

CASE STUDY: THE UNIVERSITY INFORMATION SYSTEM

In this appendix, we work out a case study, the University Information System (UnivSys.), in both frameworks by following the transformation rules and guidelines described in Chapter 4 (Pair-wise Framework) and Chapter 5 (Common Representation Framework). This case study includes the UML diagrams of the UnivSys. dealing with seminar enrollment service for a university. In this system, seminars can be created, scheduled, opened, enrolled, and closed; students can enroll in a seminar and drop a seminar if a certain requirement is met. All UML diagrams of this case study come from Amber's book [29] with minor modifications (see Figure A.1, Figure A.2 and Figure A.3). The comparison data concerning the UnivSys. in Chapter 6 (Comparison of the Two Frameworks) is calculated based on the results in this appendix.

The structure of this appendix is as follows: Section A.1 presents the results of generating each UML diagram in the PWF; Section A.2 is about CRF where we first convert UML diagrams to CGs and then present the results of generating each UML diagram from the CGs Reservoir. Note that, in a section that generates a certain UML diagram, the assumption we make is that the UML diagram to be generated

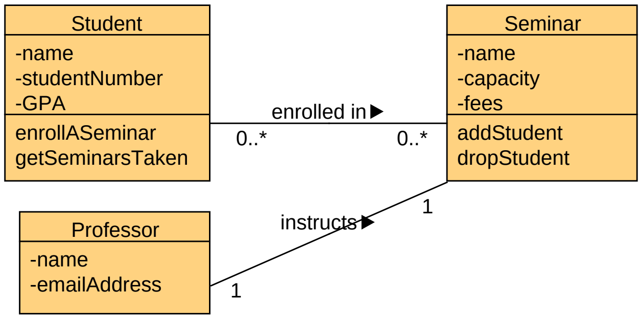


Figure A.1: UnivSys. class diagram

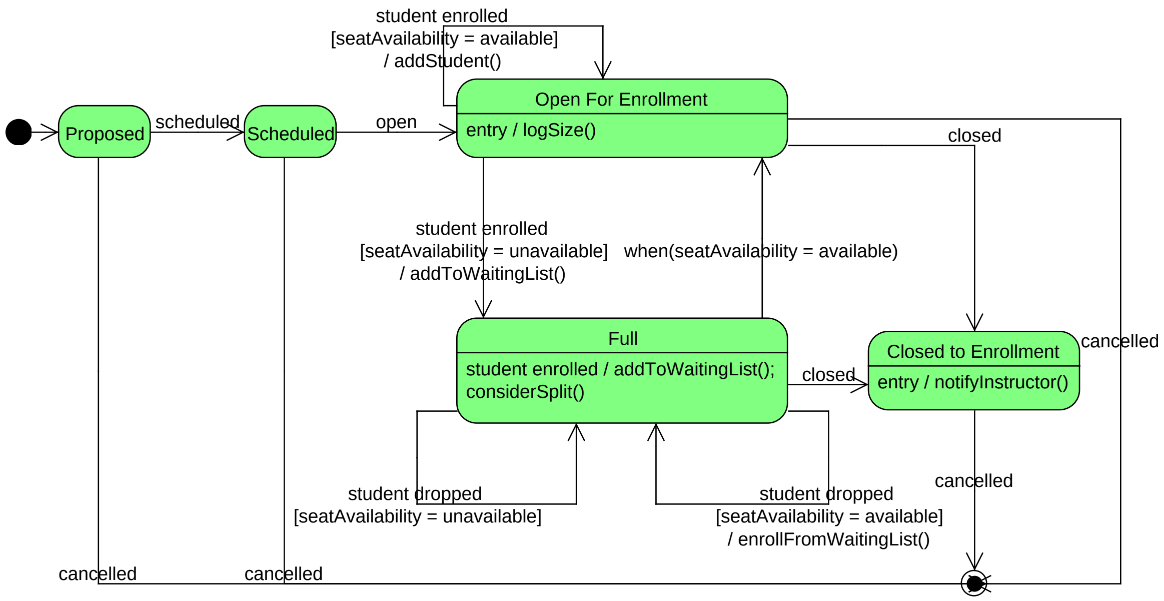


Figure A.2: `Seminar`'s state diagram

is currently not available while the other two UML diagrams are already developed (see Section 3.3 for more details). For example, when generating class diagrams in either PWF or CRF, we assume that the class diagrams of the system are missing; state diagrams and sequence diagrams then are used to generate class diagrams from scratch.

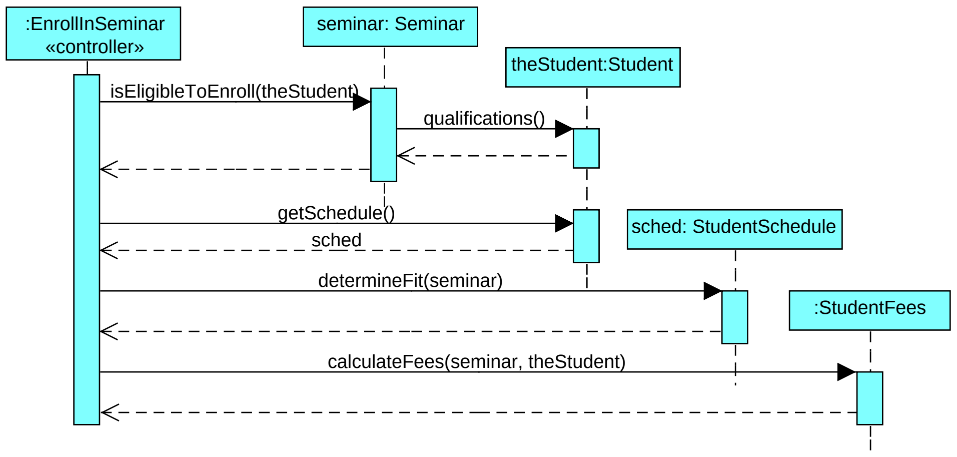


Figure A.3: UnivSys. sequence diagram

The structure of this appendix is identical to Appendix B and Appendix C, which describe the other two complete case studies from different application domains.

A.1 Pair-wise Framework

In this section, based on the pair-wise transformation rules and guidelines that are described in Chapter 4, state and sequence diagrams are transformed to class diagrams in Section A.1.1; class and sequence diagrams are transformed to state diagrams in Section A.1.2; class and state diagrams are transformed to sequence diagrams in Section A.1.3.

A.1.1 Generation of Class Diagrams

Transformation rules defined in Table 4.1 and Table 4.2 are considered to generate class diagrams. By applying the rules, the state diagram (Figure A.2) and the sequence diagram (Figure A.3) of the UnivSys. are transformed to class diagrams.

The generated class diagram with semantic holes is shown in Figure A.4.

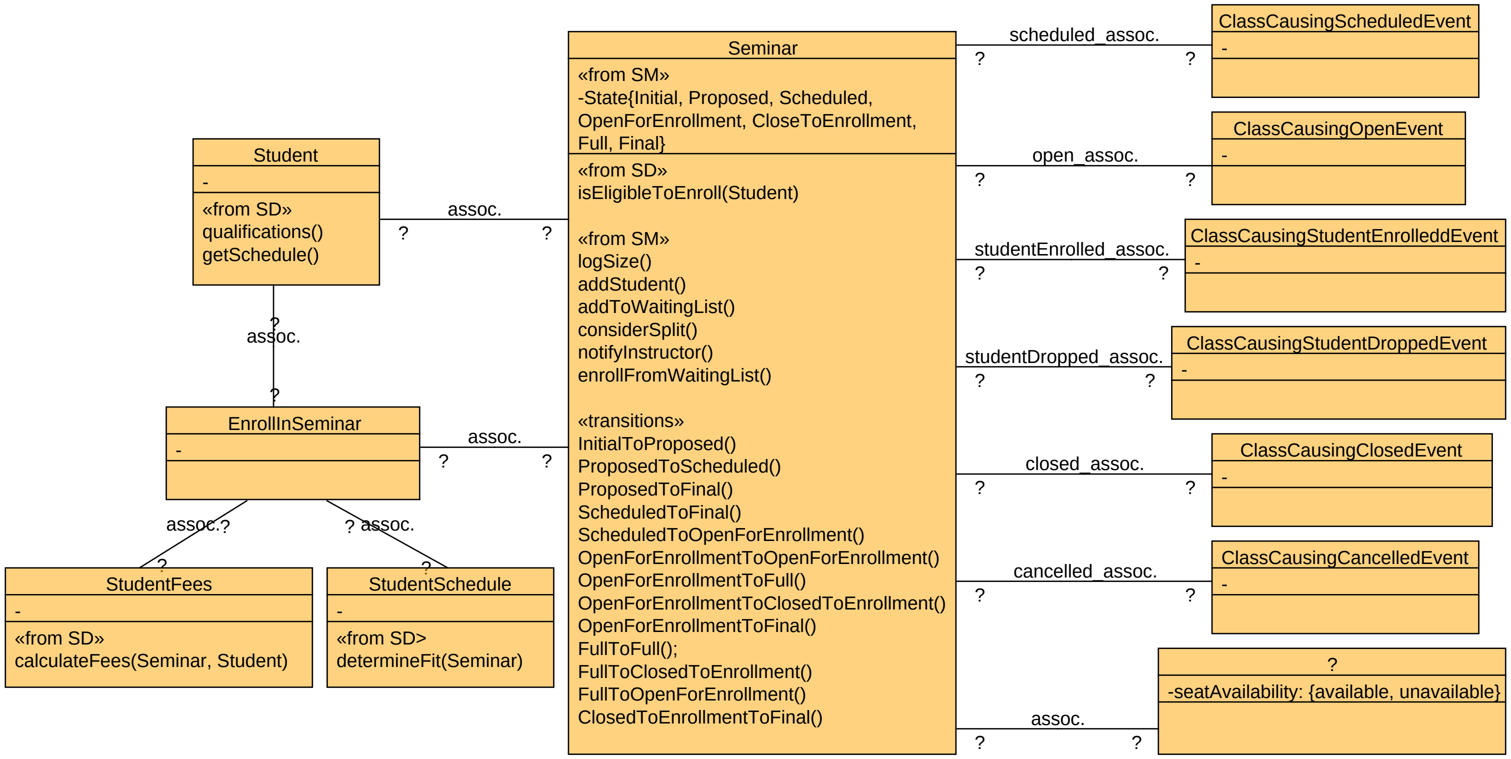

Seminar
«from SM»
-State{Initial, Proposed, Scheduled, OpenForEnrollment, CloseToEnrollment, Full, Final}
«from SD»
isEligibleToEnroll(Student)
«from SM»
logSize()
addStudent()
addToWaitingList()
considerSplit()
notifyInstructor()
enrollFromWaitingList()
«transitions»
InitialToProposed()
ProposedToScheduled()
ProposedToFinal()
ScheduledToOpenForEnrollment()
OpenForEnrollmentToOpenForEnrollment()
OpenForEnrollmentToFull()
OpenForEnrollmentToClosedToEnrollment()
OpenForEnrollmentToFinal()
FullToFull();
FullToClosedToEnrollment()
FullToOpenForEnrollment()
ClosedToEnrollmentToFinal()
Student
-
«from SD»
qualifications()
getSchedule()
EnrollInSeminar
-
StudentFees
-
«from SD»
calculateFees(Seminar, Student)
StudentSchedule
-
«from SD>
determineFit(Seminar)
ClassCausingScheduledEvent
-
ClassCausingOpenEvent
-
ClassCausingStudentEnrolleddEvent
-
ClassCausingStudentDroppedEvent
-
ClassCausingClosedEvent
-
ClassCausingCancelledEvent
-
?
-seatAvailability: {available, unavailable}
scheduled_assoc.
open_assoc.
studentEnrolled_assoc.
studentDropped_assoc.
closed_assoc.
cancelled_assoc.
assoc.
?

Figure A.4: UnivSys. generated class diagram in PWF

A.1.2 Generation of State Diagrams

Transformation rules defined in Table 4.3 are considered to generate state diagrams. By applying the rules, class diagram (Figure A.1) and sequence diagram (Figure A.3) of the UnivSys. are transformed to state diagrams. The generated state diagrams with semantic holes are shown in Figure A.5 and Figure A.6. Note that five state machines are generated: they are `EnrollInSeminar`'s state machine; `Seminar`'s state machine; `Student`'s state machine; `StudentSchedule`'s state machine and `StudentFees`'s state machine.

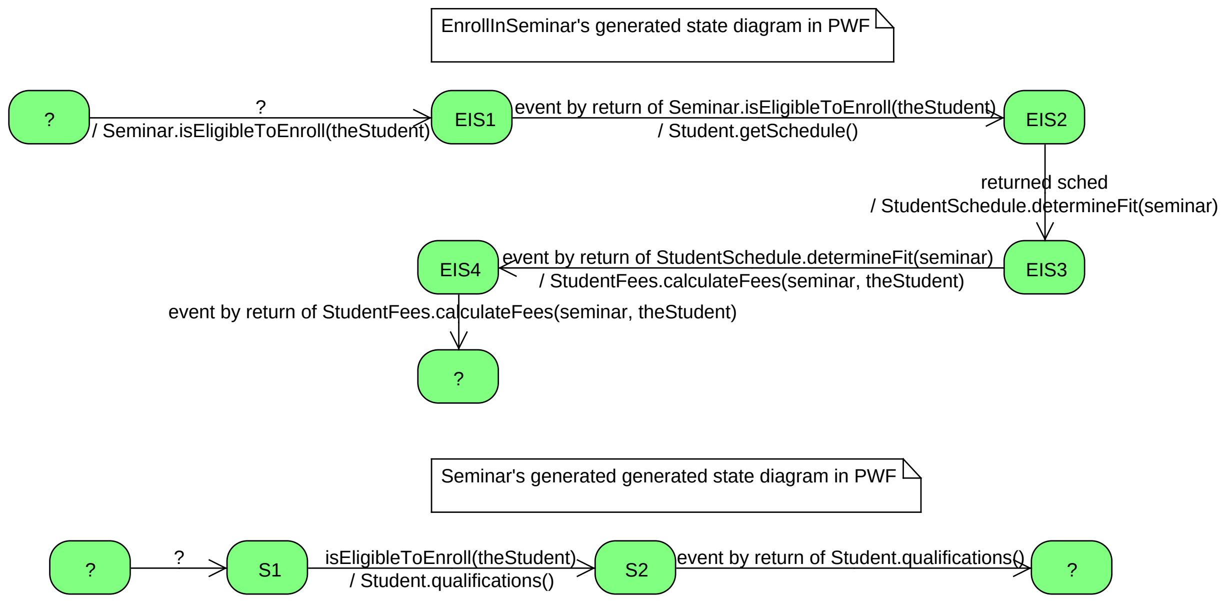

EnrollInSeminar's generated state diagram in PWF
?
?
/ Seminar.isEligibleToEnroll(theStudent)
EIS1
event by return of Seminar.isEligibleToEnroll(theStudent)
/ Student.getSchedule()
EIS2
returned sched
/ StudentSchedule.determineFit(seminar)
EIS3
event by return of StudentSchedule.determineFit(seminar)
/ StudentFees.calculateFees(seminar, theStudent)
EIS4
event by return of StudentFees.calculateFees(seminar, theStudent)
?
Seminar's generated generated state diagram in PWF
?
?
S1
isEligibleToEnroll(theStudent)
/ Student.qualifications()
S2
event by return of Student.qualifications()
?

Figure A.5: UnivSys. generated state diagram in PWF part 1

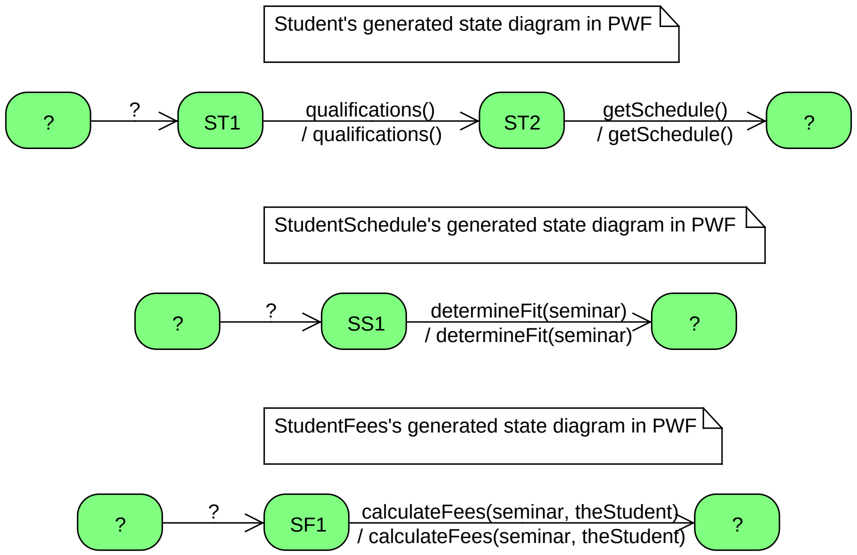

Student's generated state diagram in PWF
?
?
ST1
qualifications()
/ qualifications()
ST2
getSchedule()
/ getSchedule()
?
StudentSchedule's generated state diagram in PWF
?
?
SS1
determineFit(seminar)
/ determineFit(seminar)
?
StudentFees's generated state diagram in PWF
?
?
SF1
calculateFees(seminar, theStudent)
/ calculateFees(seminar, theStudent)
?

Figure A.6: UnivSys. generated state diagram in PWF part 2

A.1.3 Generation of Sequence Diagrams

Transformation strategies defined in Section 4.6 are considered to generate sequence diagrams. By following the transformation guidelines, the state diagram (Figure A.2) of the UnivSys. is transformed to a set of sequence diagrams. The generated sequence diagrams with semantic holes are shown in Figure A.7 to Figure A.11.

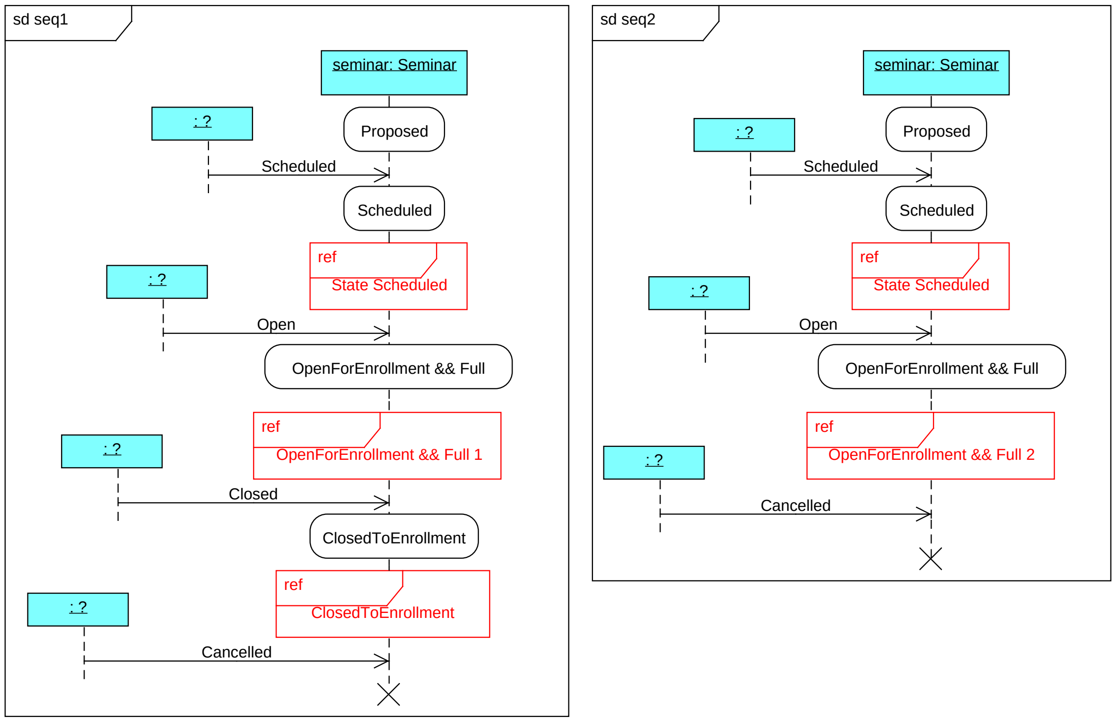


Figure A.7: UnivSys. generated sequence diagrams in PWF part 1

Figure A.7 and Figure A.8 show the four first-level sequence diagrams which represent the four main state transition paths that collectively cover all transitions in a state diagram. The set of generated sequence diagrams from a state diagram possesses the following **five properties**:

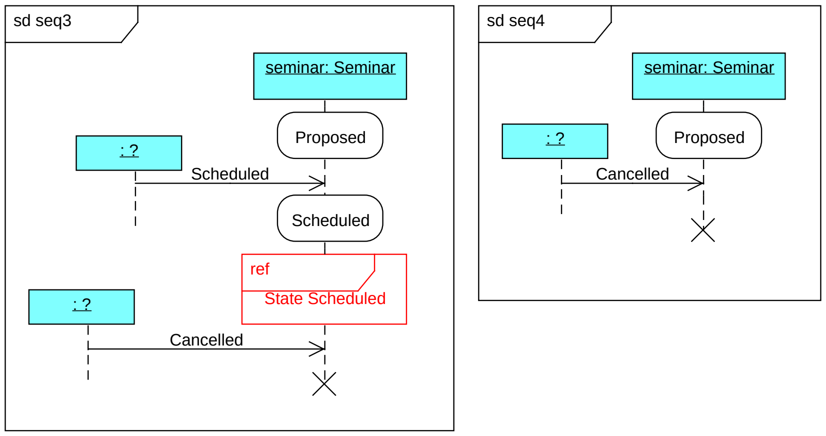


Figure A.8: UnivSys. generated sequence diagrams in PWF part 2

1. Each sequence diagram in the set starts with the initial state (the object is created) and ends in a final state (the object is destroyed);

2. Every state transition in the state diagram is covered at least once in the set of generated sequence diagrams;

3. The number of sequence diagrams is minimal given that property 2 is satisfied;

4. Each sequence diagram in the set has only one lifeline which represents the object owning the state diagram;

5. There are knowledge acquisition opportunities so the modeler is given a chance to fill in the missing requirements.

Figure A.9, Figure A.10 and Figure A.11 define separate sequences which are referenced by the first-level sequence diagrams through the `ref` fragments (interaction uses).

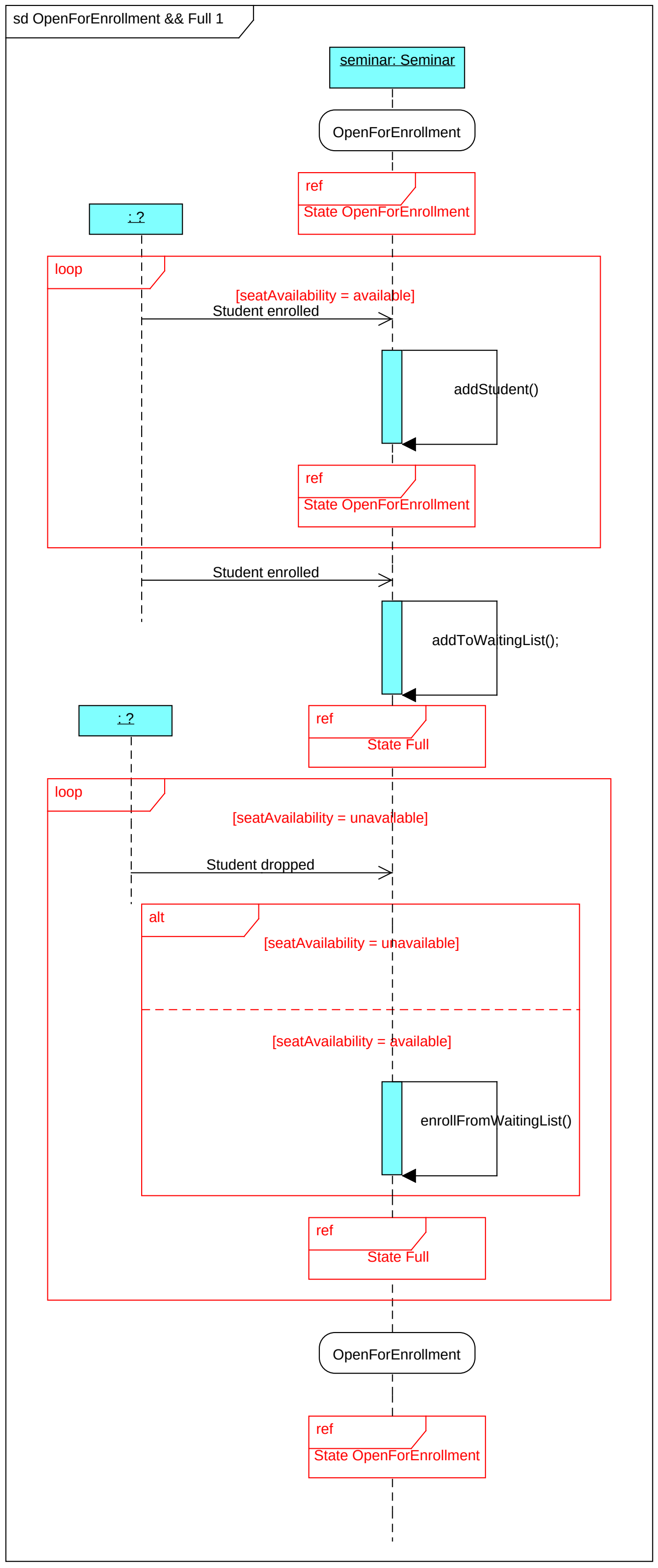


Figure A.9: UnivSys. generated sequence diagram in PWF part 3

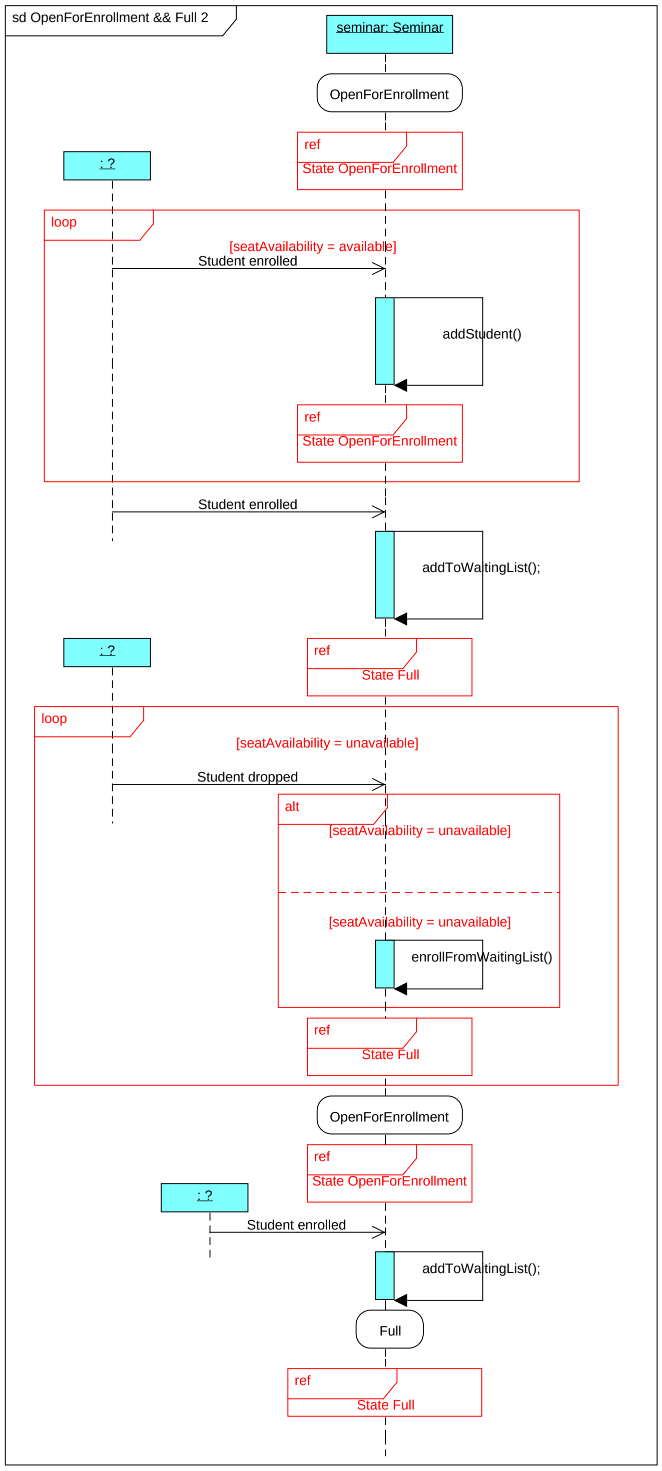


Figure A.10: UnivSys. generated sequence diagram in PWF part 4

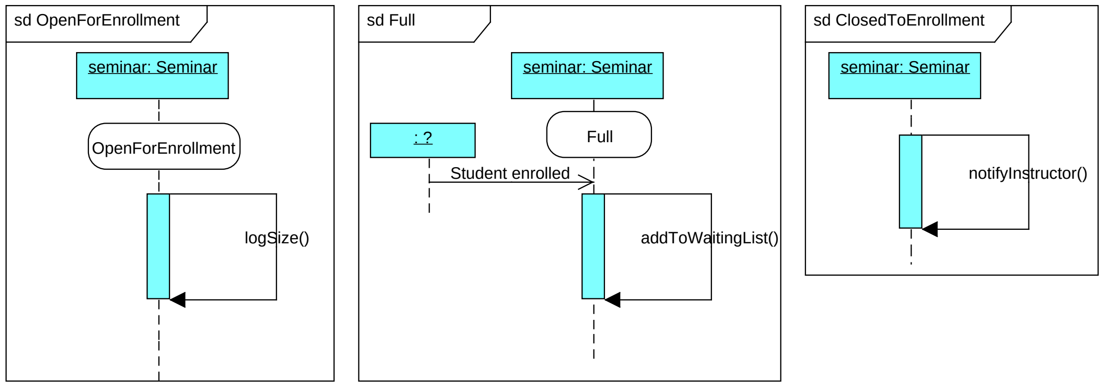


Figure A.11: UnivSys. generated sequence diagrams in PWF part 5

A.2 Common Representation Framework

In this section, we adopt a different requirements acquisition framework, the CRF, to generate UML diagrams with semantic holes for the University Information System. As detailed in Chapter 5, in order to generate the target UML diagram (e.g. state diagram), all the other UML diagrams (e.g. class and sequence diagrams) must first be converted to CGs and stored in the CGs Reservoir; inference rules of the target UML diagram are then applied to the CGs Reservoir to assert the requirements knowledge necessary to build the target UML diagram.

The results of converting class diagrams, state diagrams, and sequence diagrams of UnivSys. to CGs are shown in Section A.2.1; in Section A.1.2, each UML diagram is generated by applying corresponding inference rules to the CGs Reservoir which contains the requirements of the other two diagrams expressed in CGs form.

A.2.1 Converting UML Diagrams to CGs

Three UML diagrams of the UnivSys. (Figure A.1, Figure A.2 and Figure A.3) are converted to CGs based on the canonical graphs defined in Chapter 5. Note that the following colors are used in CGs to indicate the source of requirements knowledge: CGs in orange represent knowledge from class diagram; CGs colored green represent knowledge from state diagram; CGs colored cyan represent knowledge from sequence diagram.

A.2.1.1 Converting Class Diagrams to CGs

The canonical graphs for class diagrams are defined in Figure 5.1. By instantiating the canonical graphs, the class diagram of UnivSys. (Figure A.1) is converted to CGs (Figure A.12).

A.2.1.2 Converting State Diagrams to CGs

State transitions are organized in a state transition tree when a state diagram is converted to CGs (Figure A.13). Each concept represents a state transition and is further elaborated in a CG context in the following figures. Transitions are related by *after*.

The canonical graph for each state transition is defined in Figure 5.4. By instantiating the canonical graph for each of the 14 transitions in the state diagram of UnivSys. (Figure A.2), the CGs representing those state transitions are shown in Figure A.14 to Figure A.20. Note that each figure only shows CGs for two state transitions because of the size of each CG.

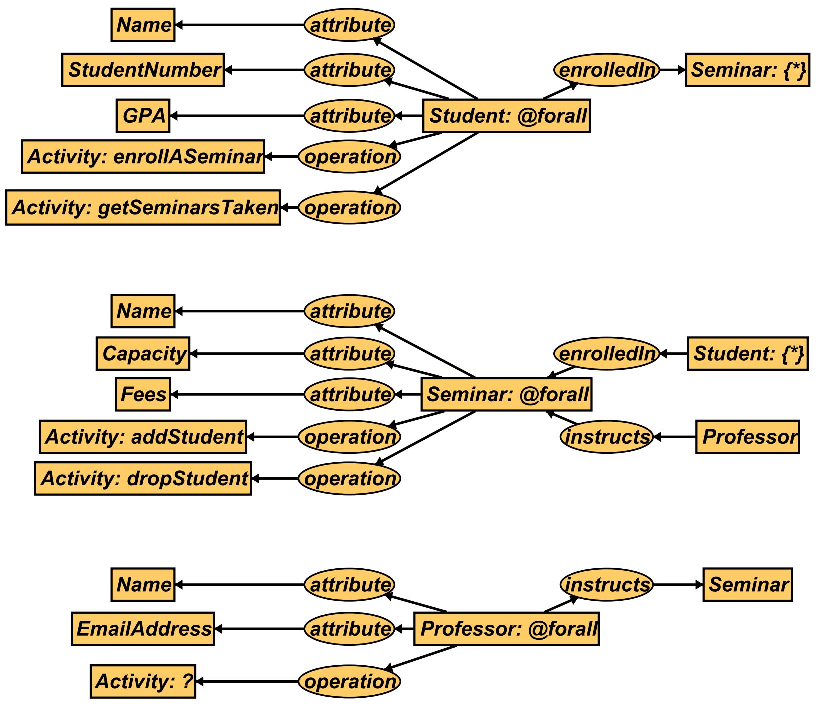


Figure A.12: UnivSys. class diagram in CGs

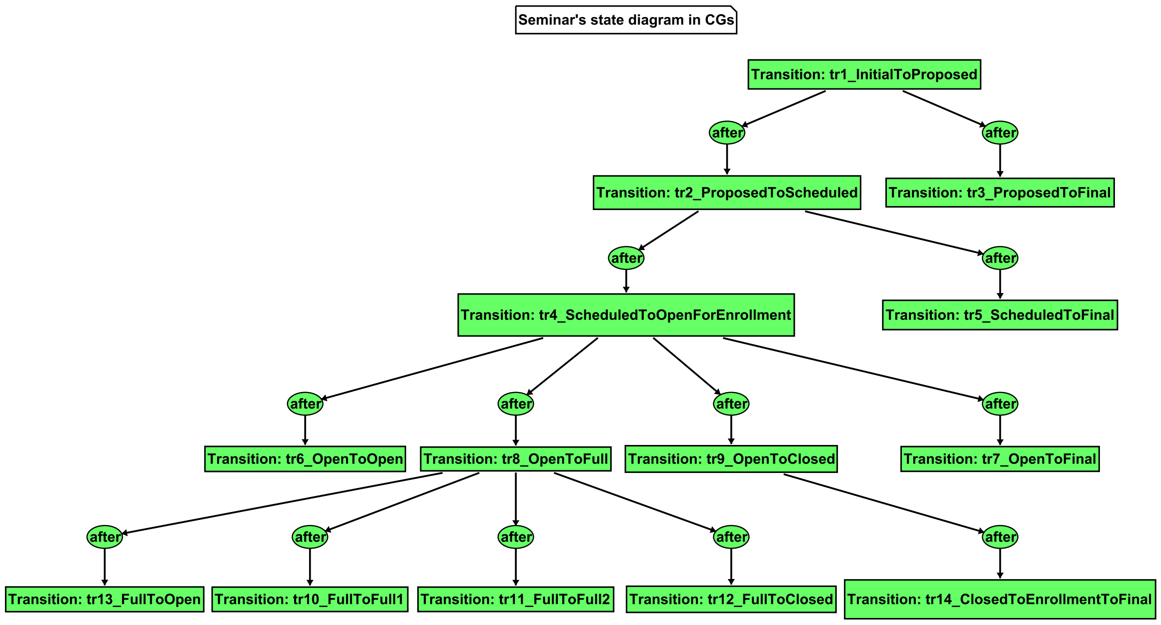
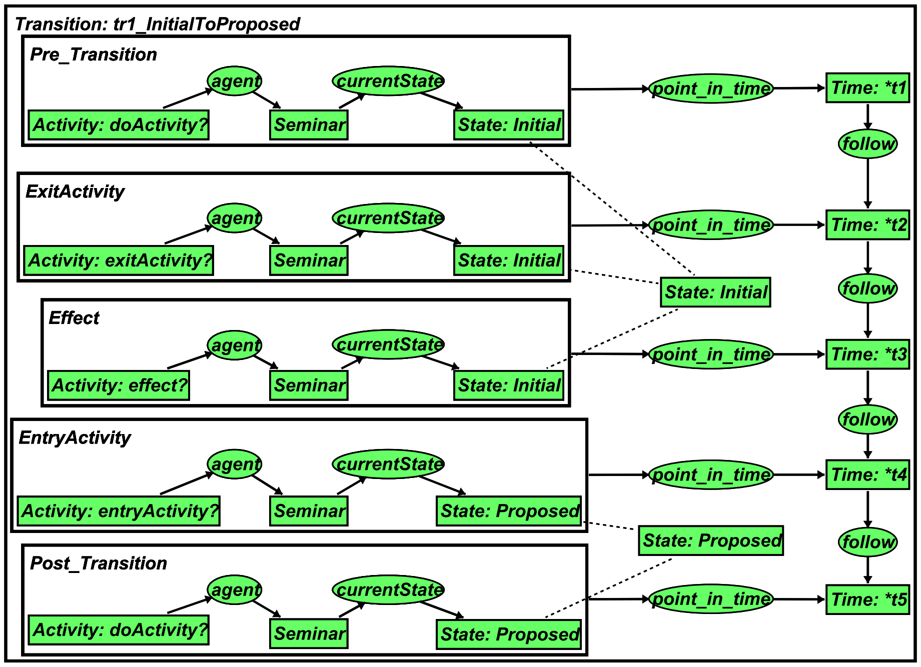

Seminar's state diagram in CGs
Transition: tr1_InitialToProposed
after
after
Transition: tr2_ProposedToScheduled
Transition: tr3_ProposedToFinal
after
after
Transition: tr4_ScheduledToOpenForEnrollment
Transition: tr5_ScheduledToFinal
after
after
after
after
Transition: tr6_OpenToOpen
Transition: tr8_OpenToFull
Transition: tr9_OpenToClosed
Transition: tr7_OpenToFinal
after
after
after
after
after
Transition: tr13_FullToOpen
Transition: tr10_FullToFull1
Transition: tr11_FullToFull2
Transition: tr12_FullToClosed
Transition: tr14_ClosedToEnrollmentToFinal

Figure A.13: UnivSys. state diagram in CGs



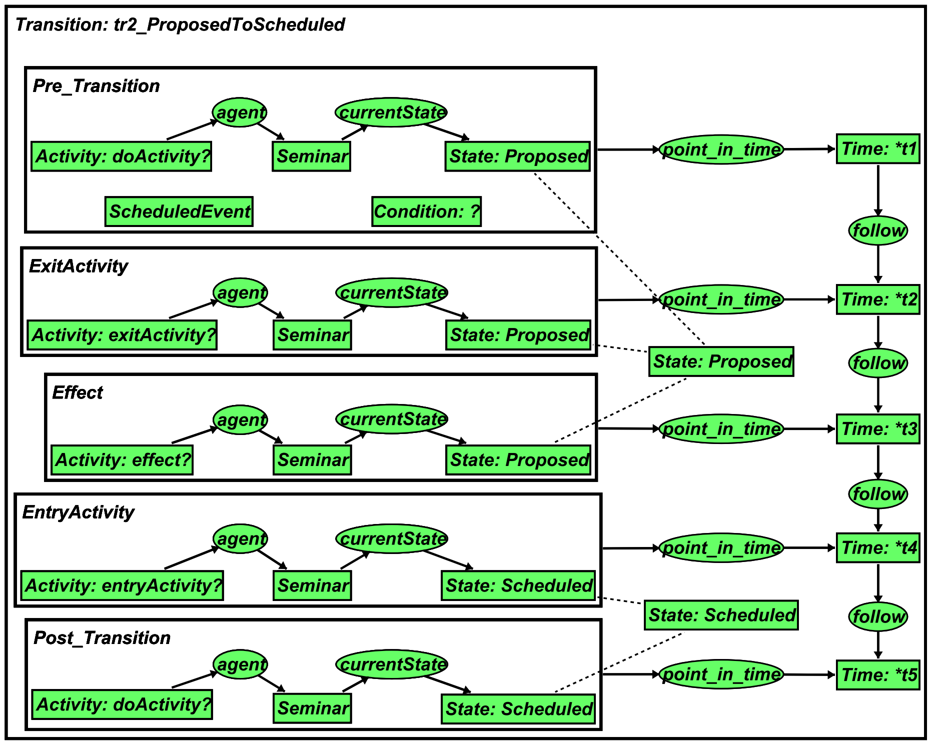


Figure A.14: UnivSys. state diagram in CGs part 1

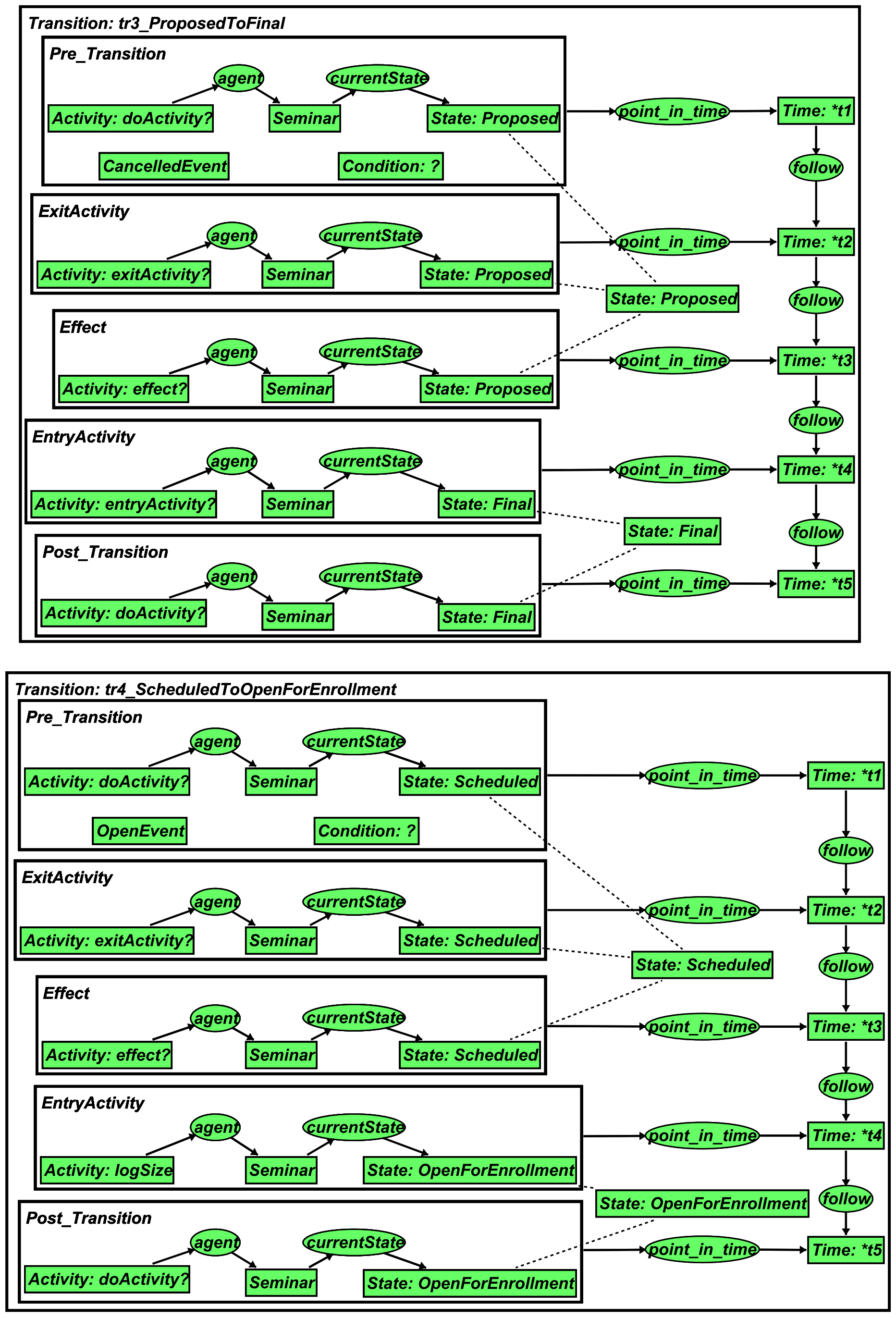


Figure A.15: UnivSys. state diagram in CGs part 2

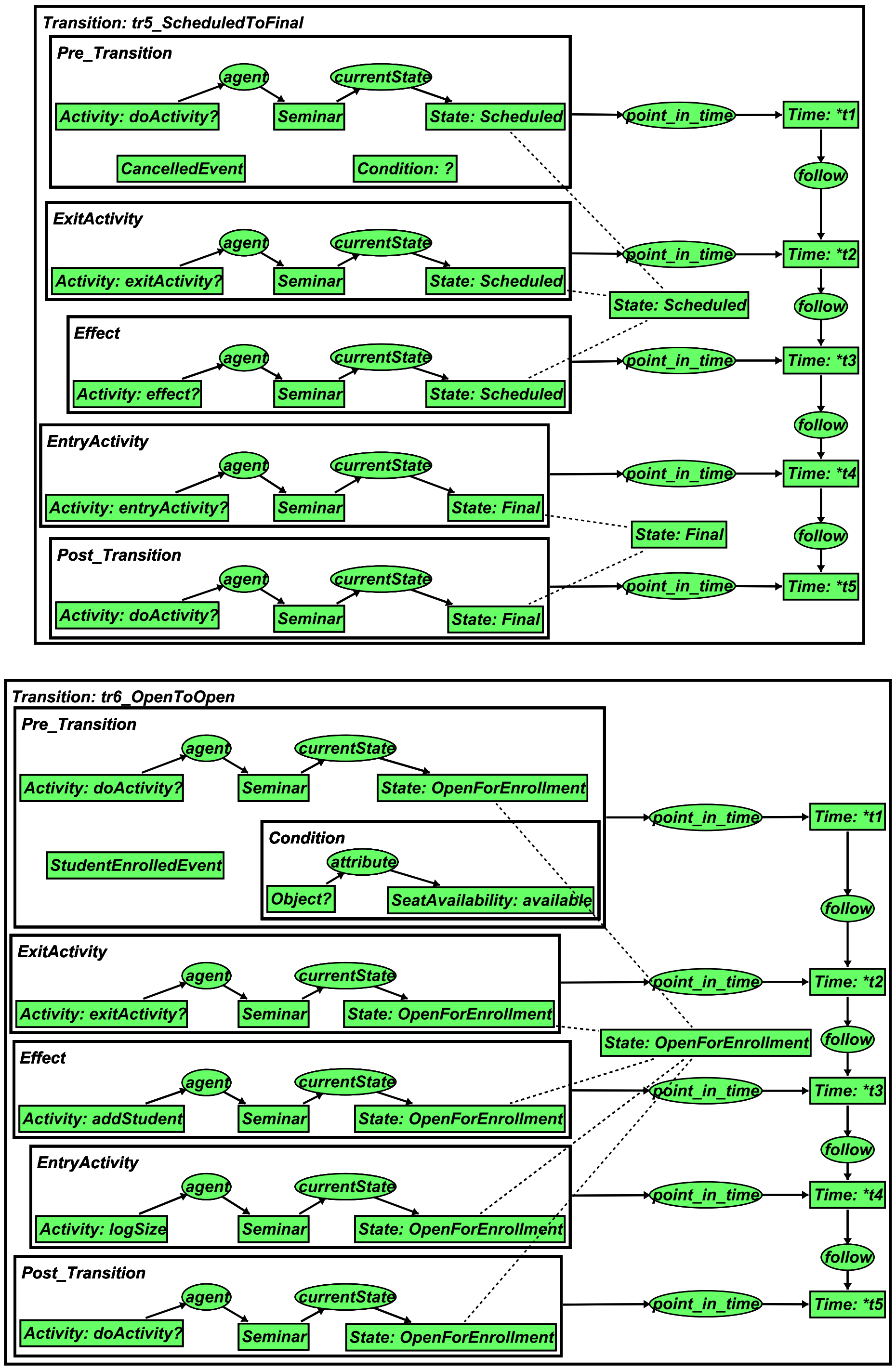


Figure A.16: UnivSys. state diagram in CGs part 3

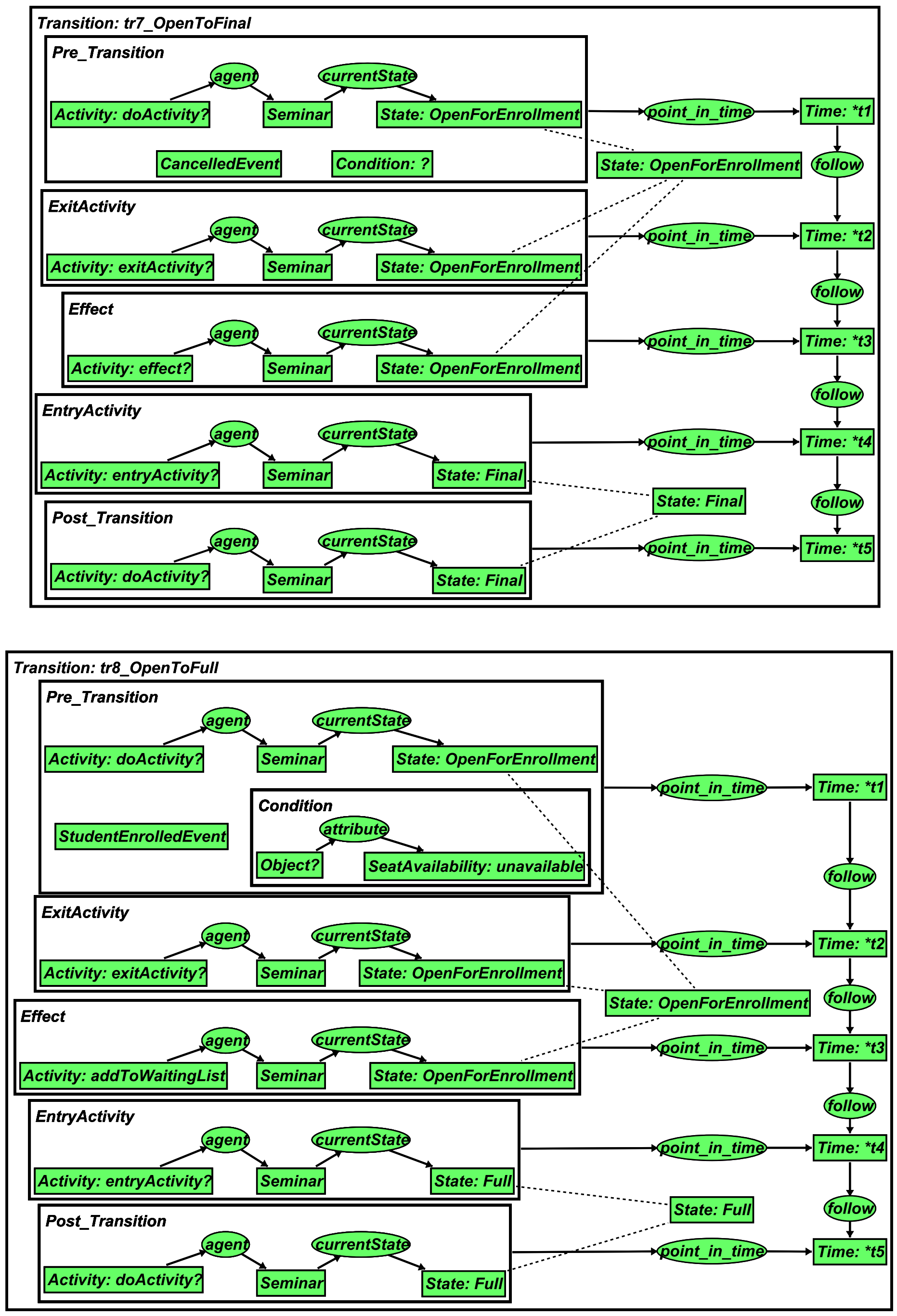
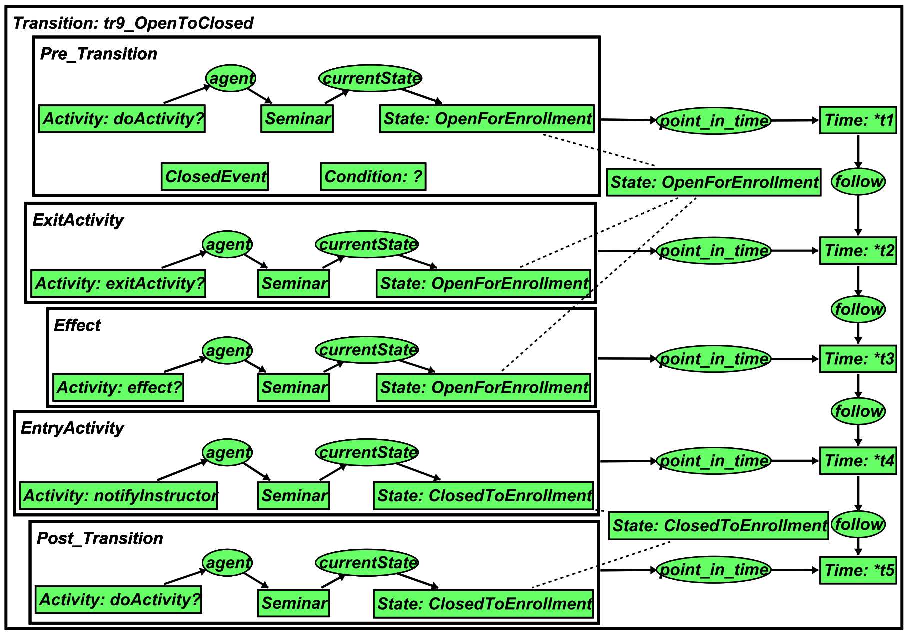


Figure A.17: UnivSys. state diagram in CGs part 4



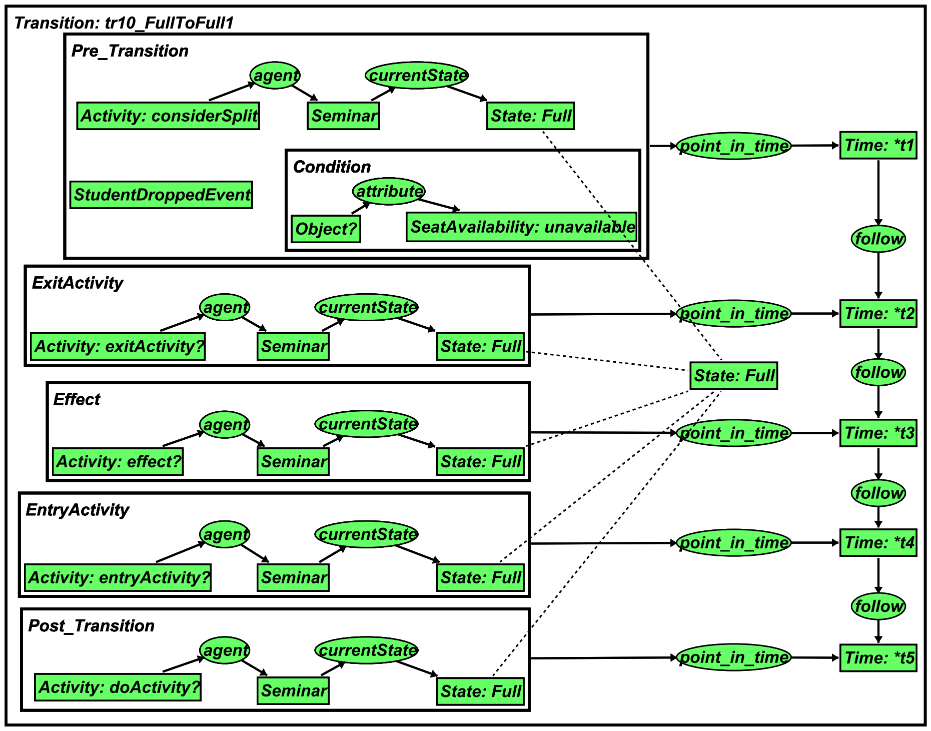


Figure A.18: UnivSys. state diagram in CGs part 5

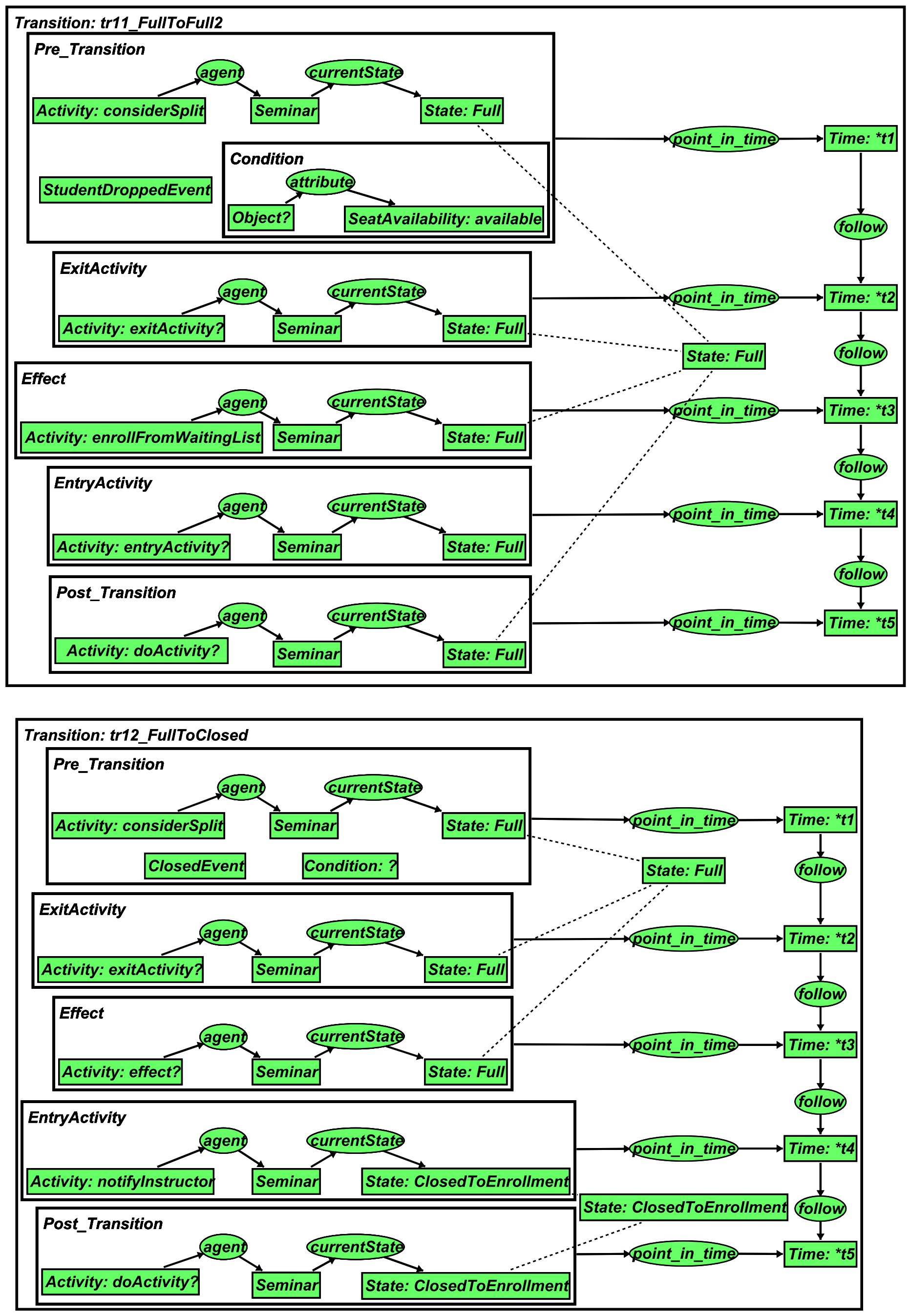
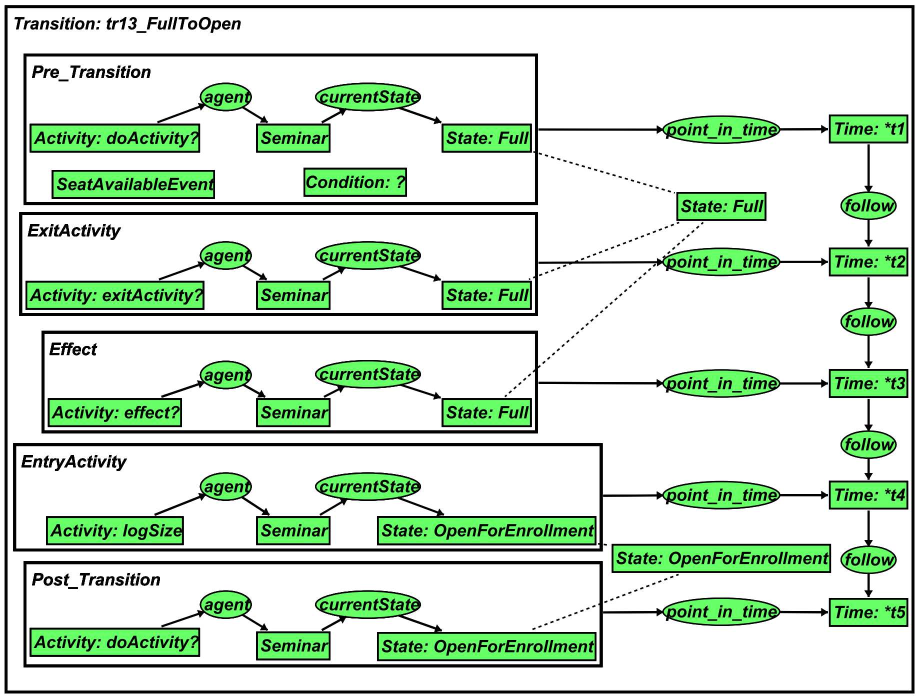


Figure A.19: UnivSys. state diagram in CGs part 6



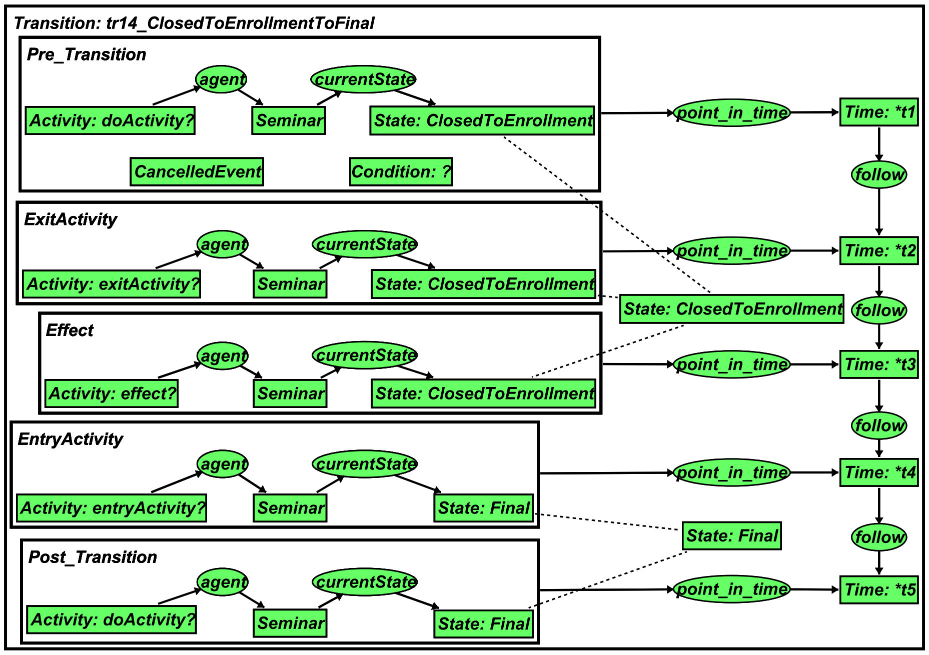


Figure A.20: UnivSys. state diagram in CGs part 7

A.2.1.3 Converting Sequence Diagrams to CGs

Sequence diagrams are made up of messages passing among objects. The canonical graph for a message passing is defined in Figure 5.10. By instantiating the canonical graphs, the sequence diagram of UnivSys. (Figure A.3) is converted to CGs. Because of the large size of the CG model of sequence diagram, it is divided into two CGs and presented in Figure A.21 and Figure A.22, respectively.

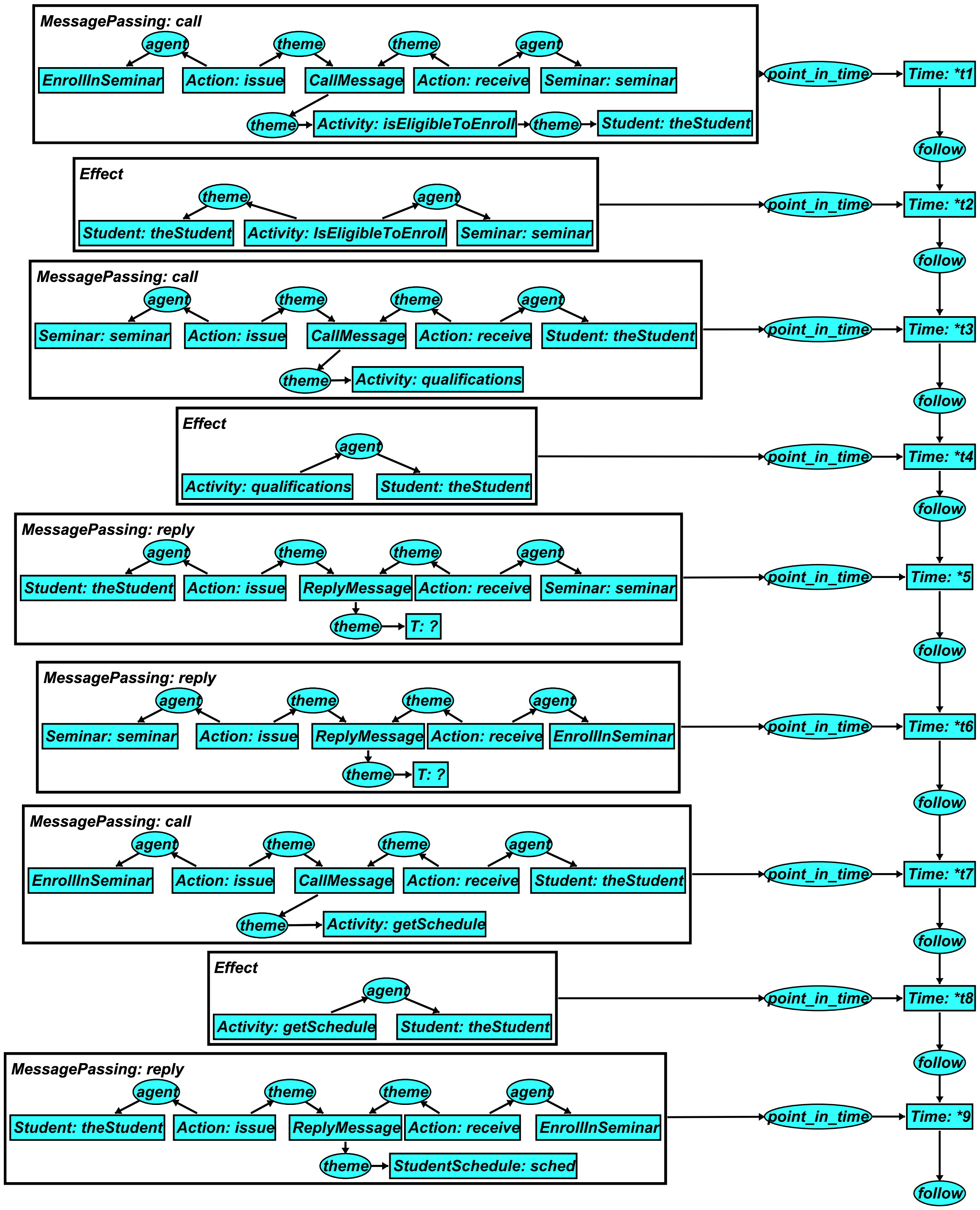


Figure A.21: UnivSys. sequence diagram in CGs part 1

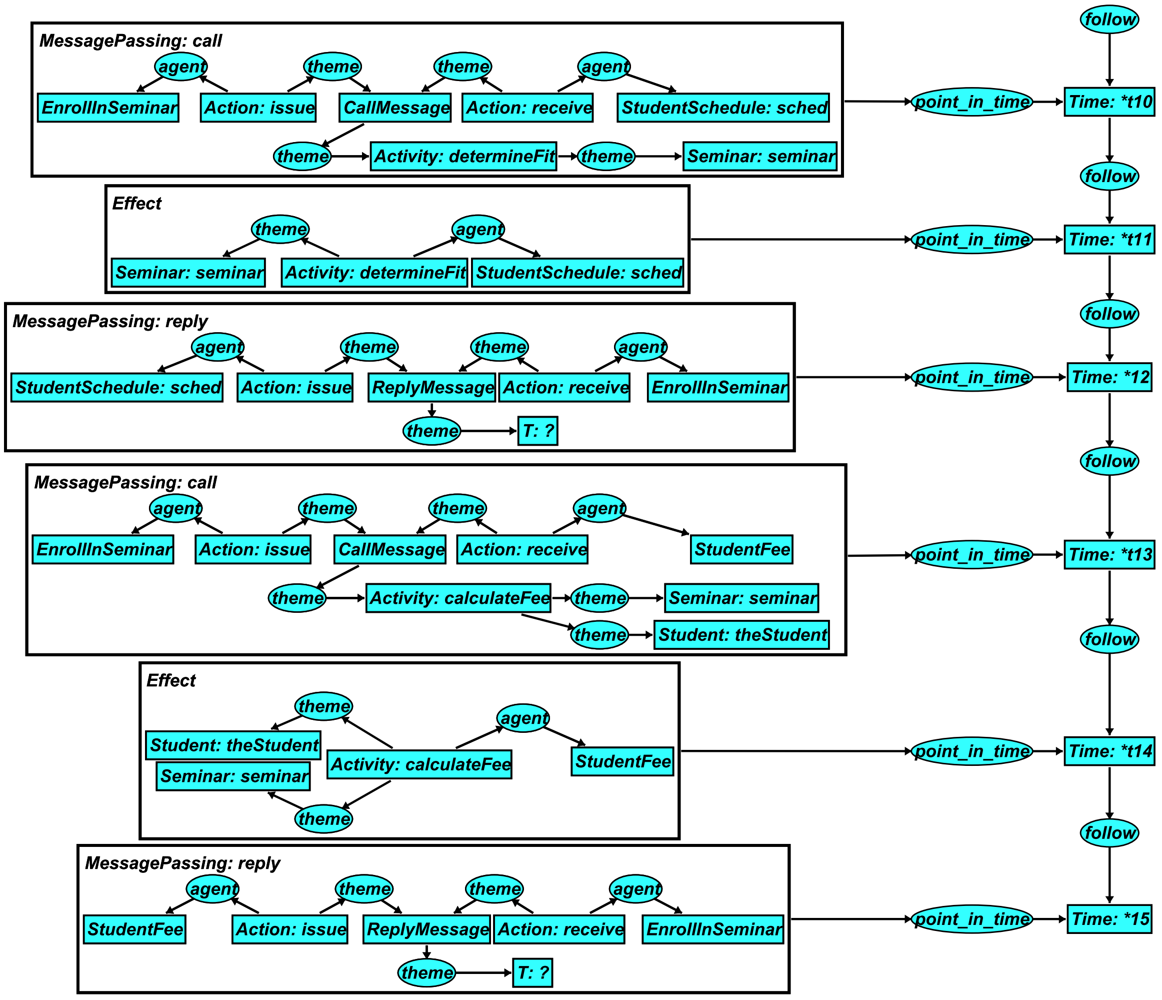


Figure A.22: UnivSys. sequence diagram in CGs part 2

A.2.2 Generating UML Diagrams with Semantic Holes from the CGs Reservoir

With the UML diagrams being expressed in CGs form in the previous section, in this section requirements knowledge needed to build target UML diagrams for UnivSys. is inferred according to the inference rules. Note that CGs in light gray represent requirements knowledge inferred by inference rules.

A.2.2.1 Generating Class Diagrams from the CGs Reservoir

In this subsection, the class diagram is the target UML diagram to be built and the state diagram and sequence diagram of UnivSys. have already been converted to CGs and stored in CGs Reservoir (see their CGs in Section A.2.1.2 and Section A.2.1.3). The inference rules for generating class models are defined in Figure 5.15 and Figure 5.16 in Chapter 5. Eleven classes are inferred. The results of the generated CGs are shown in Figure A.23 to Figure A.25.

Then the CGs are translated back to the UML class diagram notations (Figure A.26).

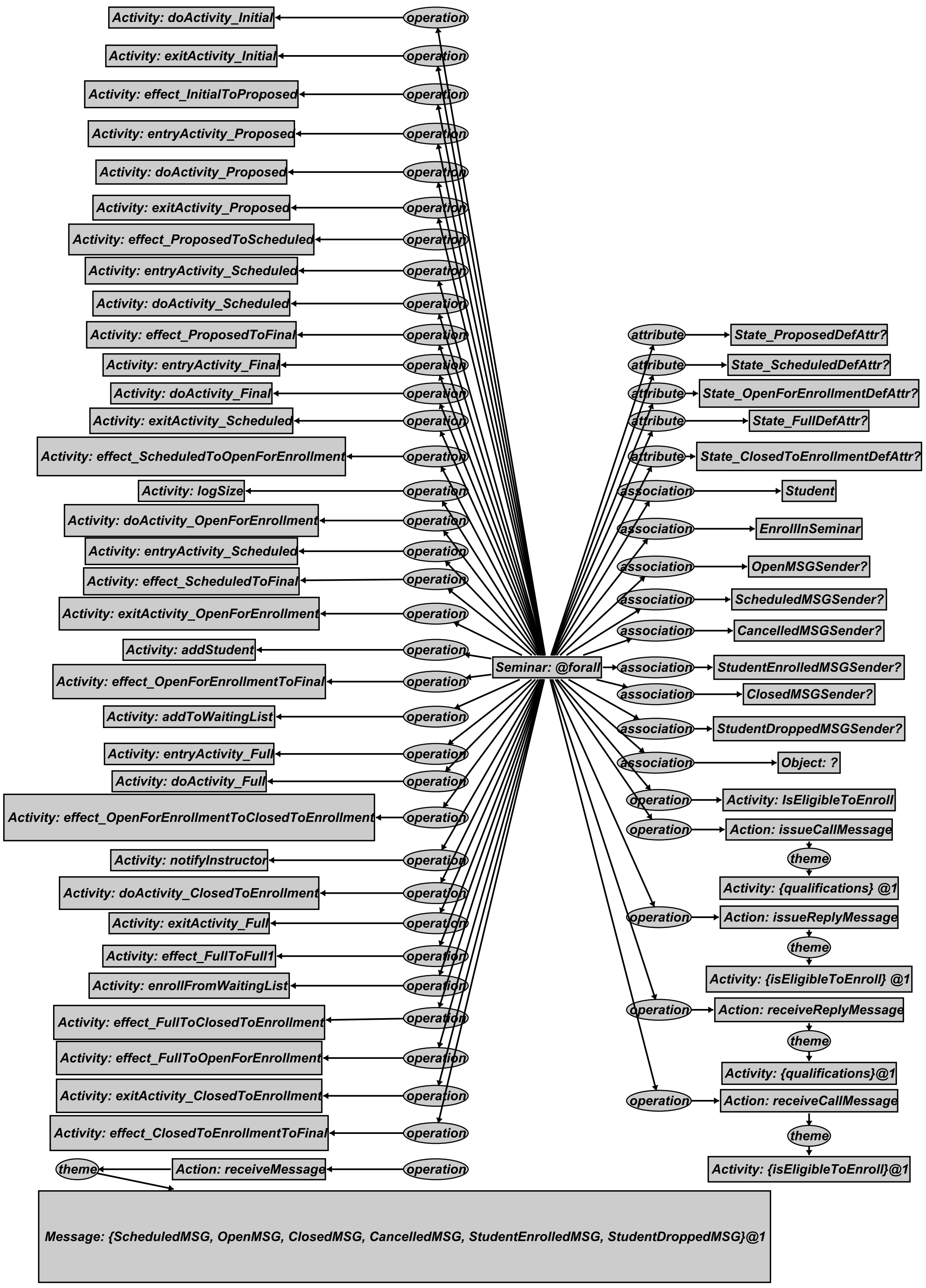

Seminar: @forall
operation
Activity: doActivity_Initial
Activity: exitActivity_Initial
Activity: effect_InitialToProposed
Activity: entryActivity_Proposed
Activity: doActivity_Proposed
Activity: exitActivity_Proposed
Activity: effect_ProposedToScheduled
Activity: entryActivity_Scheduled
Activity: doActivity_Scheduled
Activity: effect_ProposedToFinal
Activity: entryActivity_Final
Activity: doActivity_Final
Activity: exitActivity_Scheduled
Activity: effect_ScheduledToOpenForEnrollment
Activity: logSize
Activity: doActivity_OpenForEnrollment
Activity: entryActivity_Scheduled
Activity: effect_ScheduledToFinal
Activity: exitActivity_OpenForEnrollment
Activity: addStudent
Activity: effect_OpenForEnrollmentToFinal
Activity: addToWaitingList
Activity: entryActivity_Full
Activity: doActivity_Full
Activity: effect_OpenForEnrollmentToClosedToEnrollment
Activity: notifyInstructor
Activity: doActivity_ClosedToEnrollment
Activity: exitActivity_Full
Activity: effect_FullToFull1
Activity: enrollFromWaitingList
Activity: effect_FullToClosedToEnrollment
Activity: effect_FullToOpenForEnrollment
Activity: exitActivity_ClosedToEnrollment
Activity: effect_ClosedToEnrollmentToFinal
Action: receiveMessage
theme
Message: {ScheduledMSG, OpenMSG, ClosedMSG, CancelledMSG, StudentEnrolledMSG, StudentDroppedMSG}@1
attribute
State_ProposedDefAttr?
State_ScheduledDefAttr?
State_OpenForEnrollmentDefAttr?
State_FullDefAttr?
State_ClosedToEnrollmentDefAttr?
association
Student
EnrollInSeminar
OpenMSGSender?
ScheduledMSGSender?
CancelledMSGSender?
StudentEnrolledMSGSender?
ClosedMSGSender?
StudentDroppedMSGSender?
Object: ?
Activity: IsEligibleToEnroll
Action: issueCallMessage
Activity: {qualifications} @1
Action: issueReplyMessage
Activity: {isEligibleToEnroll} @1
Action: receiveReplyMessage
Activity: {qualifications}@1
Action: receiveCallMessage
Activity: {isEligibleToEnroll}@1

Figure A.23: UnivSys. generated class model in CGs part 1

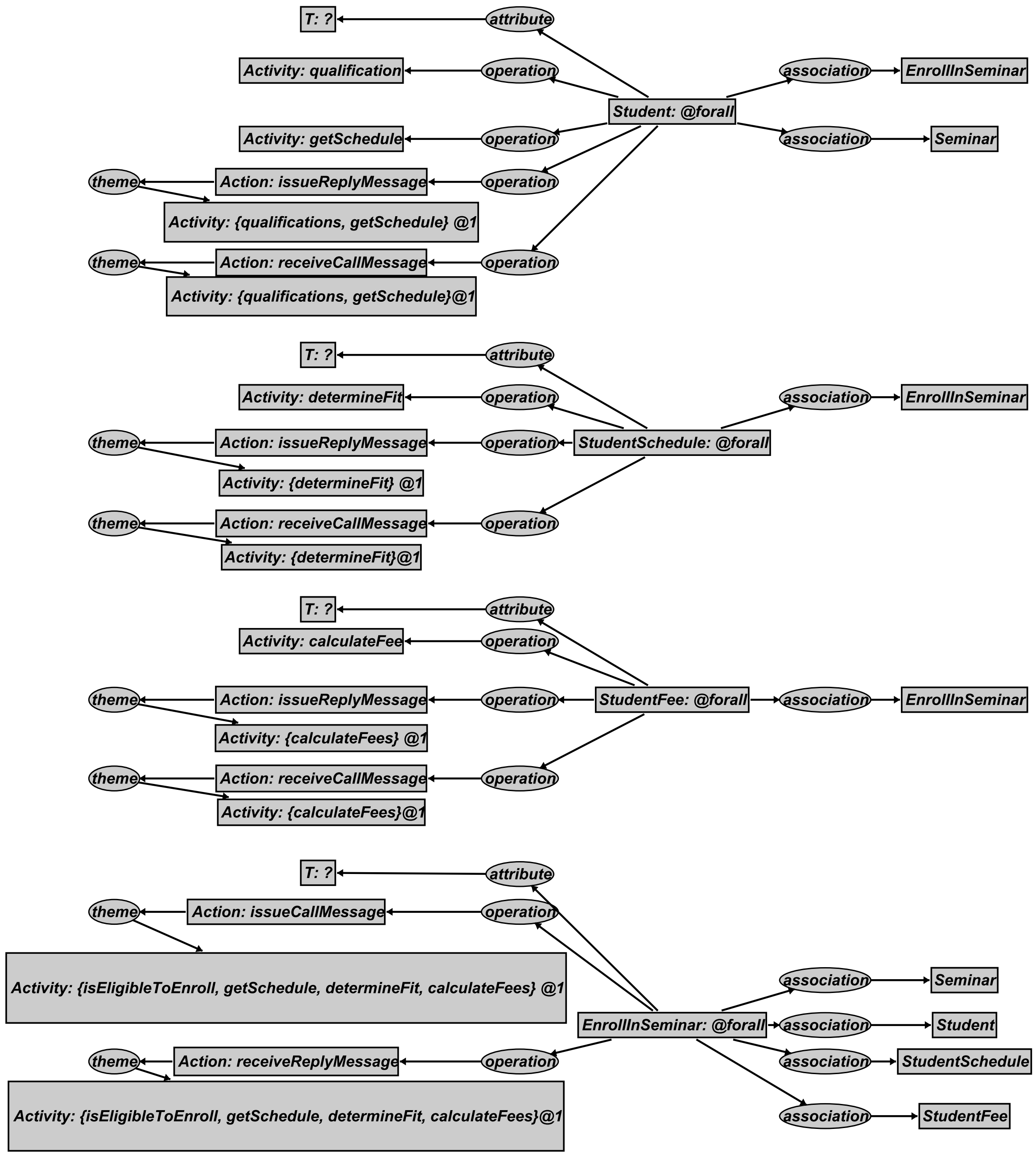


Figure A.24: UnivSys. generated class model in CGs part 2

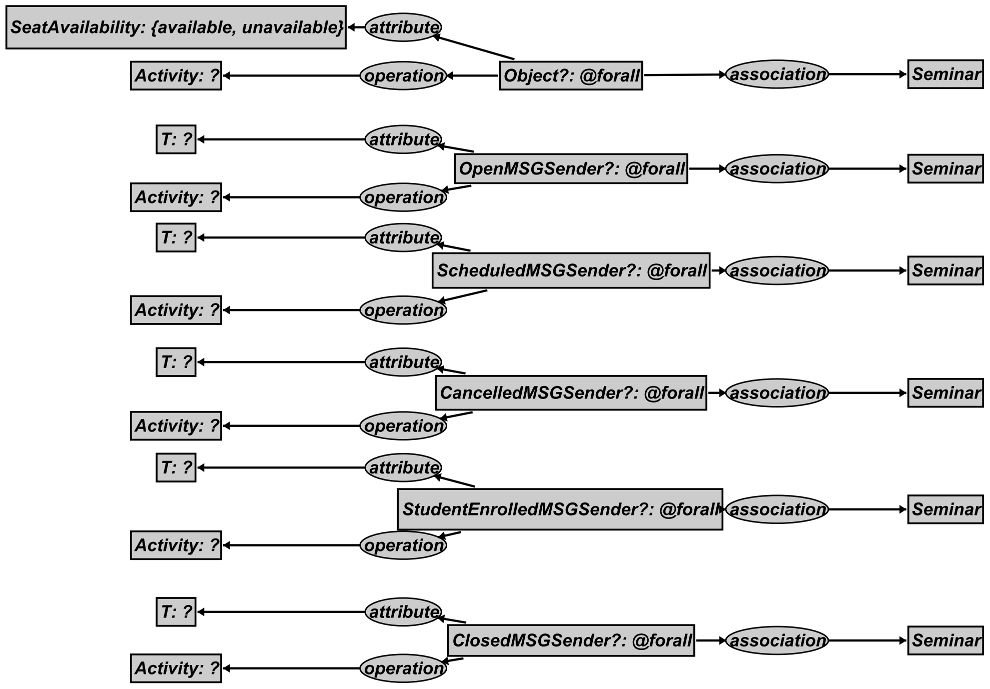


Figure A.25: UnivSys. generated class model in CGs part 3

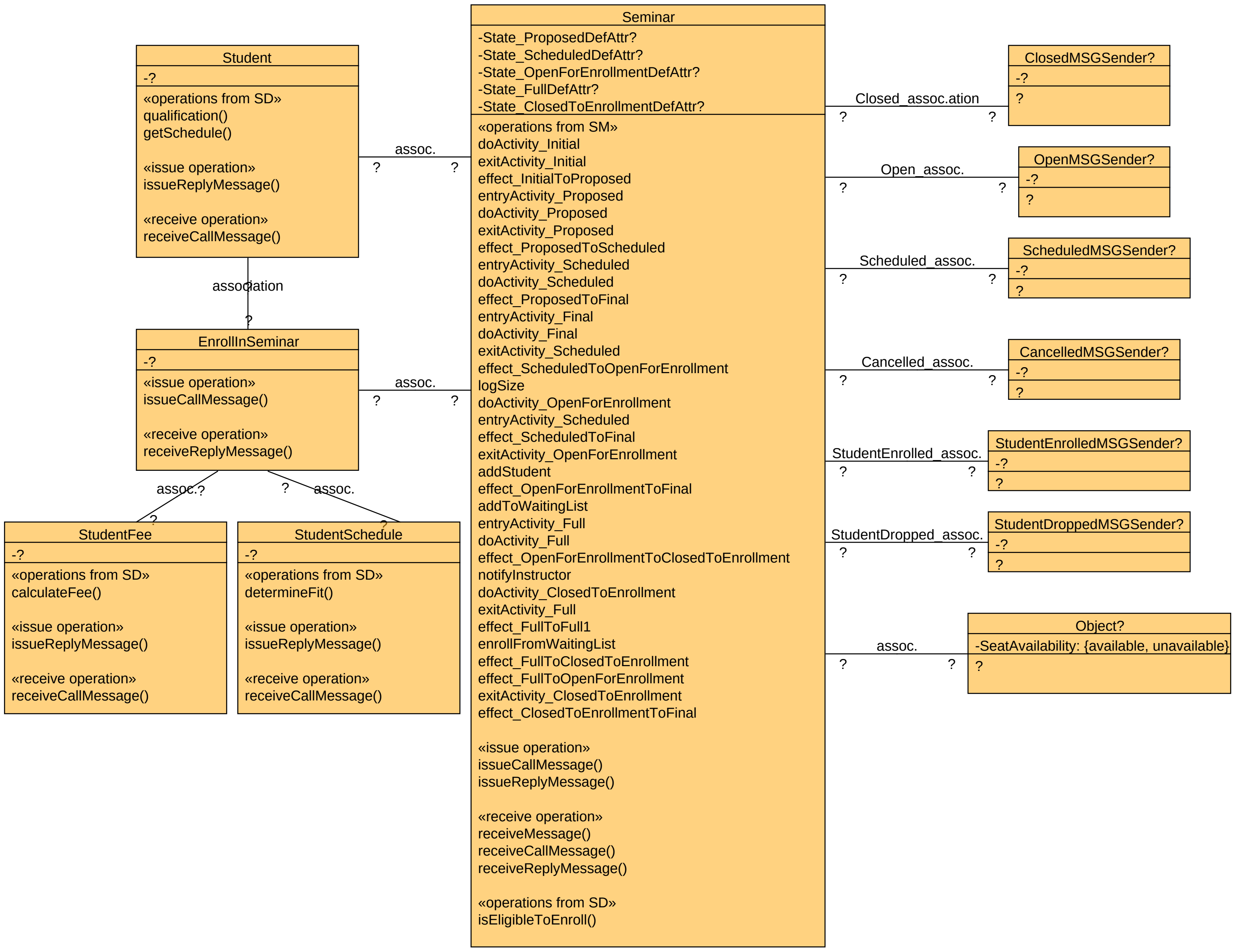


Figure A.26: UnivSys. generated class diagram in CRF

A.2.2.2 Generating State Diagrams from the CGs Reservoir

In this subsection, state diagram is the target UML diagram to be built and the class diagram and sequence diagram of UnivSys. have already been converted to CGs and stored in CGs Reservoir (see their CGs in Section A.2.1.1 and Section A.2.1.3). The inference rules for generating state diagrams are defined in Section 5.2.2.1. After the rules are applied on the CGs Reservoir, the CGs for state diagrams are generated. Because of the size of the CGs, they are shown in two separate figures, Figure A.27 and Figure A.28.

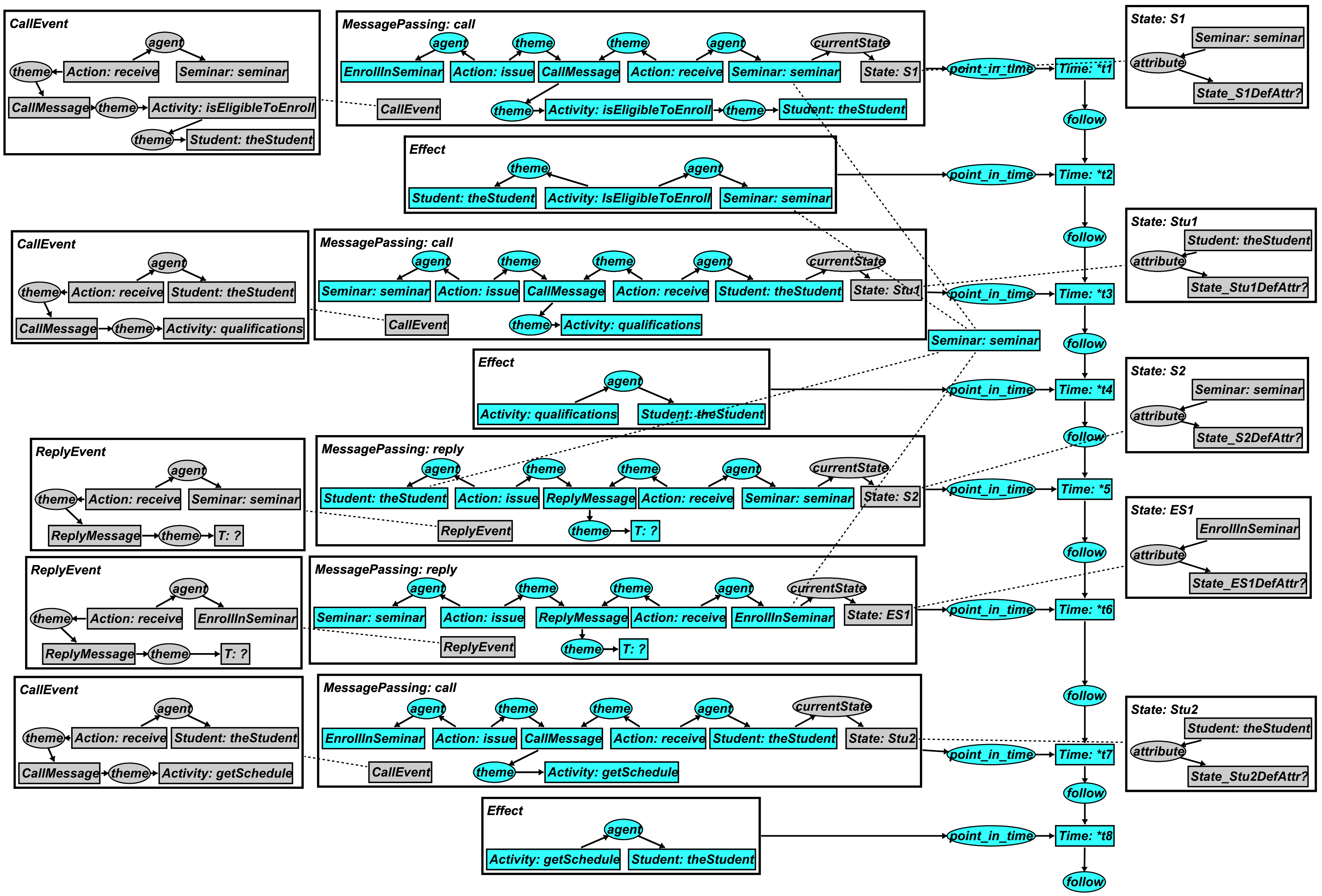


Figure A.27: UnivSys. generated state model in CGs part 1

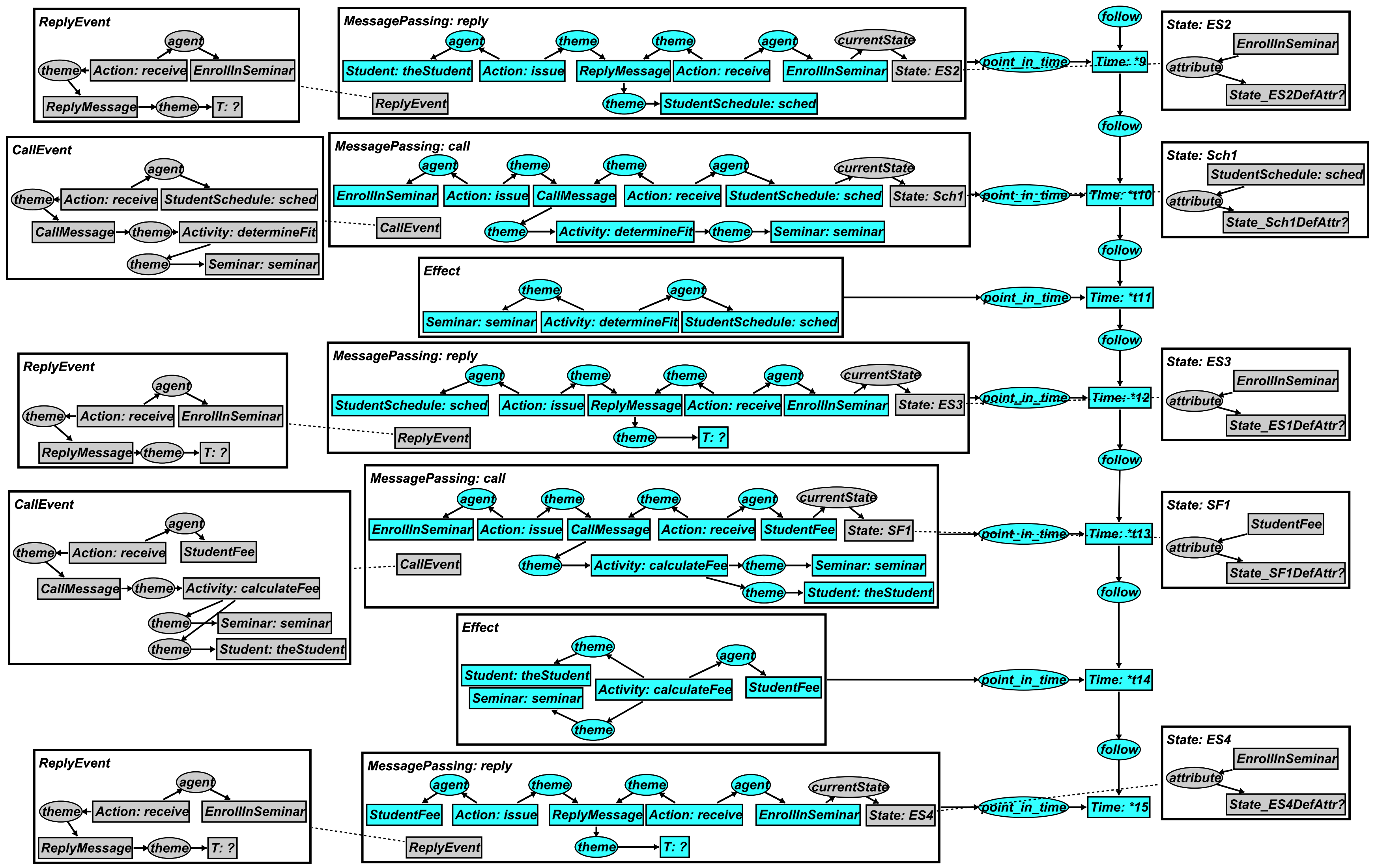


Figure A.28: UnivSys. generated state model in CGs part 2

The CGs are then translated back to the UML state diagram notations (Figure A.29 to Figure A.33). Note that five state machines are generated.

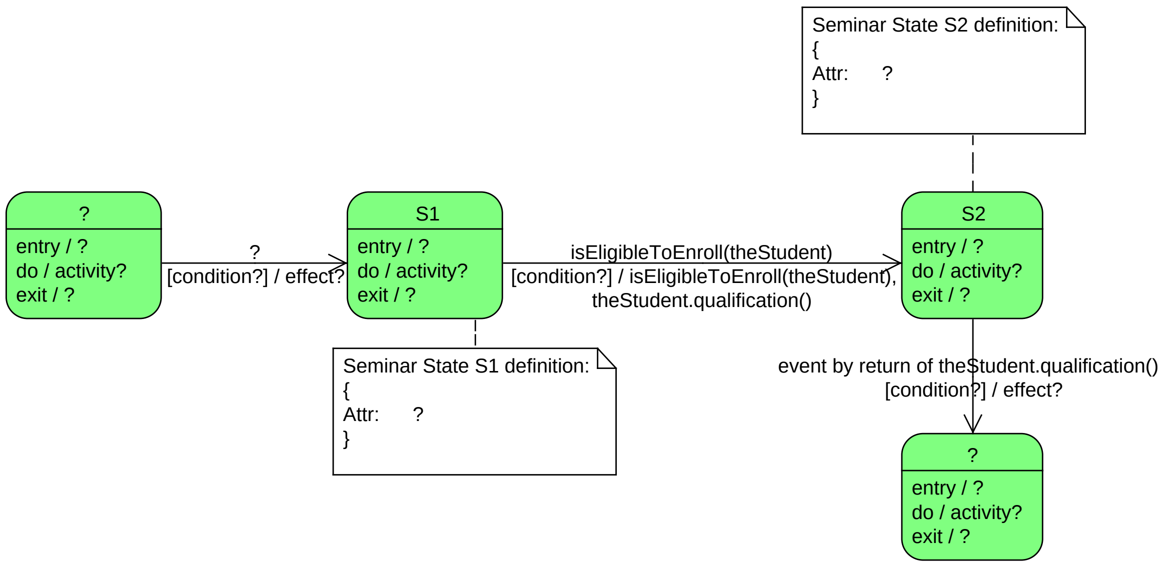

Seminar State S2 definition:
{
Attr: ?
}
?
entry / ?
do / activity?
exit / ?
?
[condition?] / effect?
S1
entry / ?
do / activity?
exit / ?
isEligibleToEnroll(theStudent)
[condition?] / isEligibleToEnroll(theStudent),
theStudent.qualification()
S2
entry / ?
do / activity?
exit / ?
Seminar State S1 definition:
{
Attr: ?
}
event by return of theStudent.qualification()
[condition?] / effect?
?
entry / ?
do / activity?
exit / ?

Figure A.29: UnivSys. generated state diagram for `Seminar` in CRF

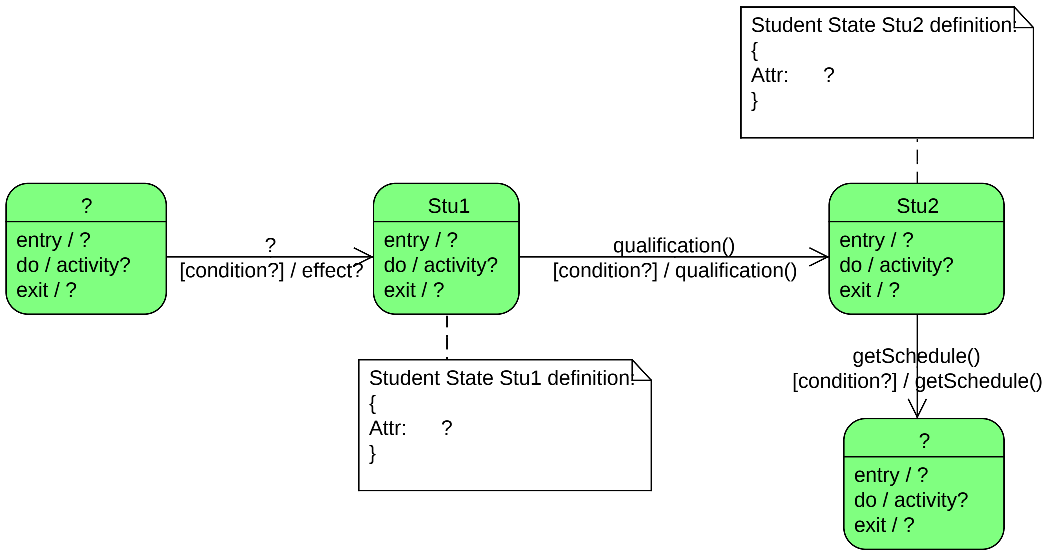


Figure A.30: UnivSys. generated state diagram for `Student` in CRF

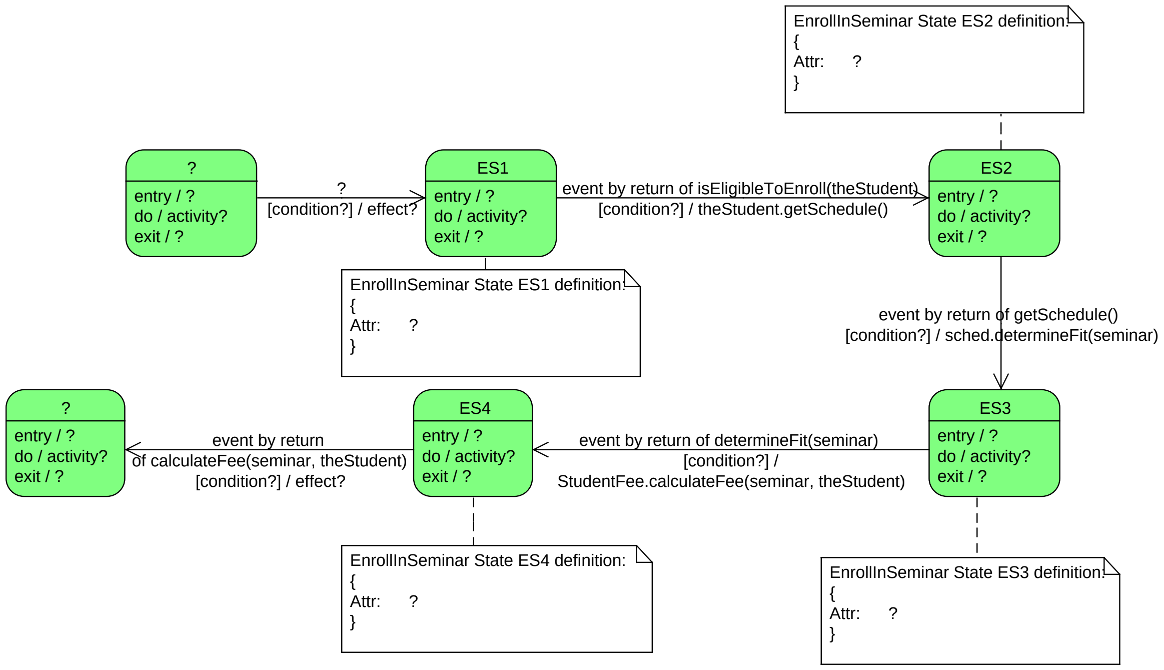

EnrollInSeminar State ES2 definition:
{
Attr: ?
}
?
entry / ?
do / activity?
exit / ?
?
[condition?] / effect?
ES1
entry / ?
do / activity?
exit / ?
event by return of isEligibleToEnroll(theStudent)
[condition?] / theStudent.getSchedule()
ES2
entry / ?
do / activity?
exit / ?
EnrollInSeminar State ES1 definition:
{
Attr: ?
}
event by return of getSchedule()
[condition?] / sched.determineFit(seminar)
?
entry / ?
do / activity?
exit / ?
event by return
of calculateFee(seminar, theStudent)
[condition?] / effect?
ES4
entry / ?
do / activity?
exit / ?
event by return of determineFit(seminar)
[condition?] /
StudentFee.calculateFee(seminar, theStudent)
ES3
entry / ?
do / activity?
exit / ?
EnrollInSeminar State ES4 definition:
{
Attr: ?
}
EnrollInSeminar State ES3 definition:
{
Attr: ?
}

Figure A.31: UnivSys. generated state diagram for `EnrollInSeminar` in CRF

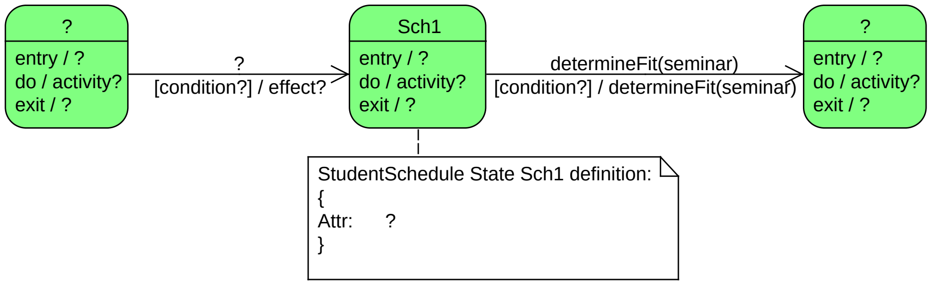


Figure A.32: UnivSys. generated state diagram for `StudentSchedule` in CRF

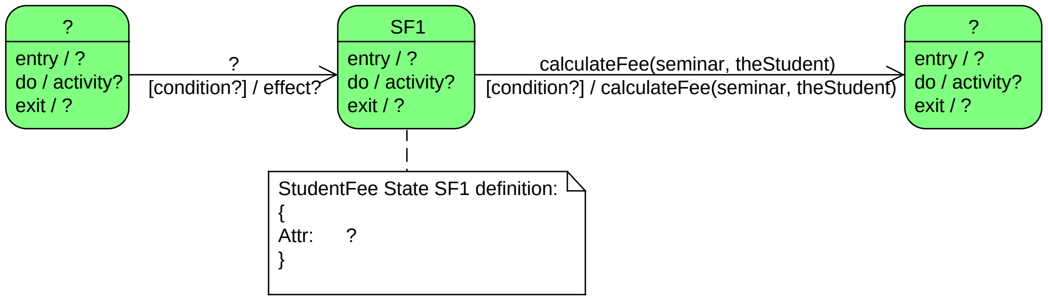


Figure A.33: UnivSys. generated state diagram for `StudentFee` in CRF

A.2.2.3 Generating Sequence Diagrams from the CGs Reservoir

In this subsection, a sequence diagram is the target UML diagram to be built and the class diagram and state diagram of UnivSys. have already been converted to CGs and stored in CGs Reservoir (see their CGs in Section A.2.1.1 and Section A.2.1.2). The inference rules for generating sequence diagrams are defined in Section 5.2.3.1. After the rules are applied on the CGs Reservoir, the CGs for sequence diagrams are generated. In this case study, nine sequences are identified in CGs and they are shown in nine separate figures, Figure A.34 to Figure A.42. Each of the sequences will result in one sequence diagram when CGs are translated back to sequence diagram notations.

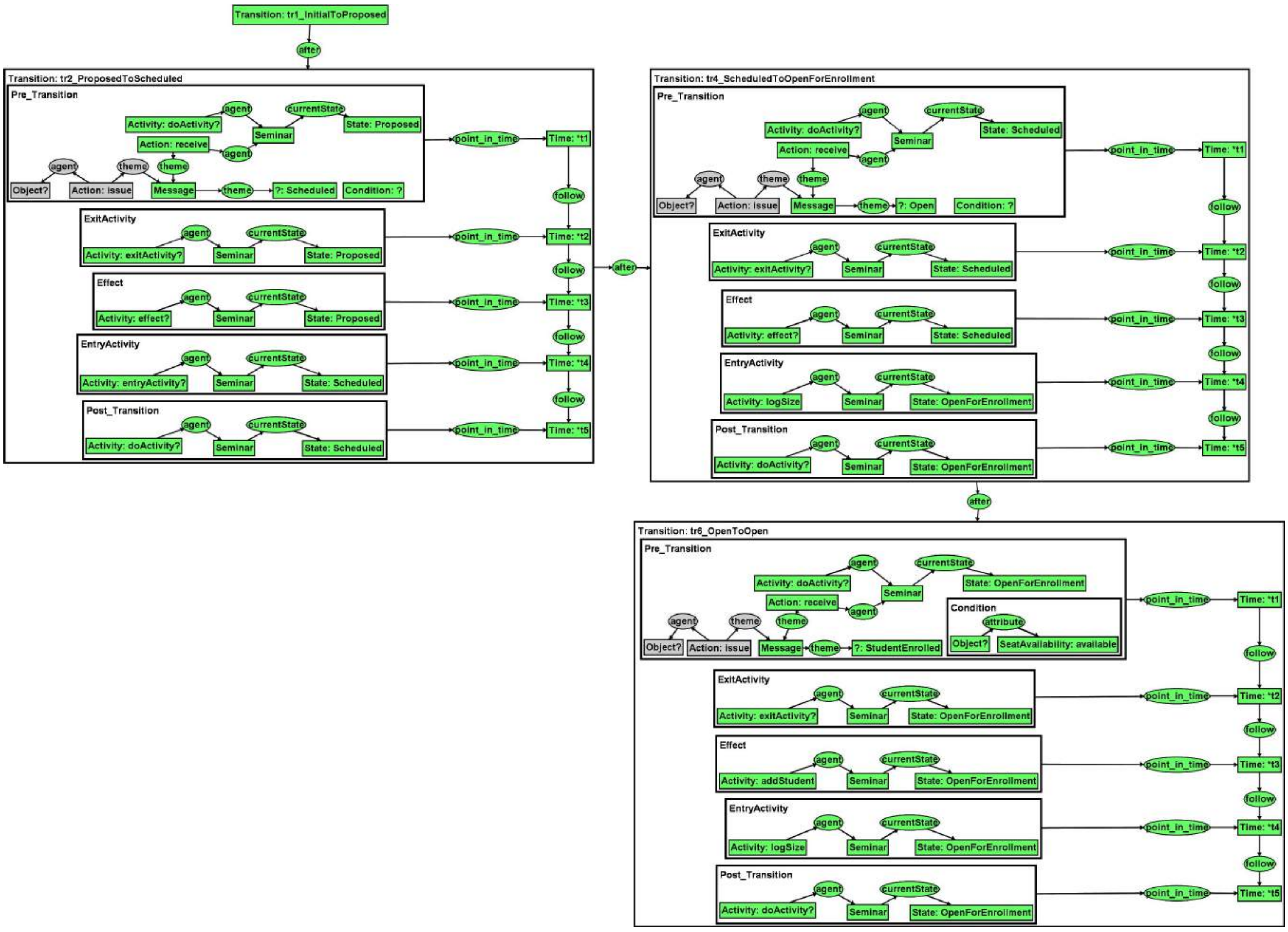


Figure A.34: UnivSys. generated sequence 1 in CGs

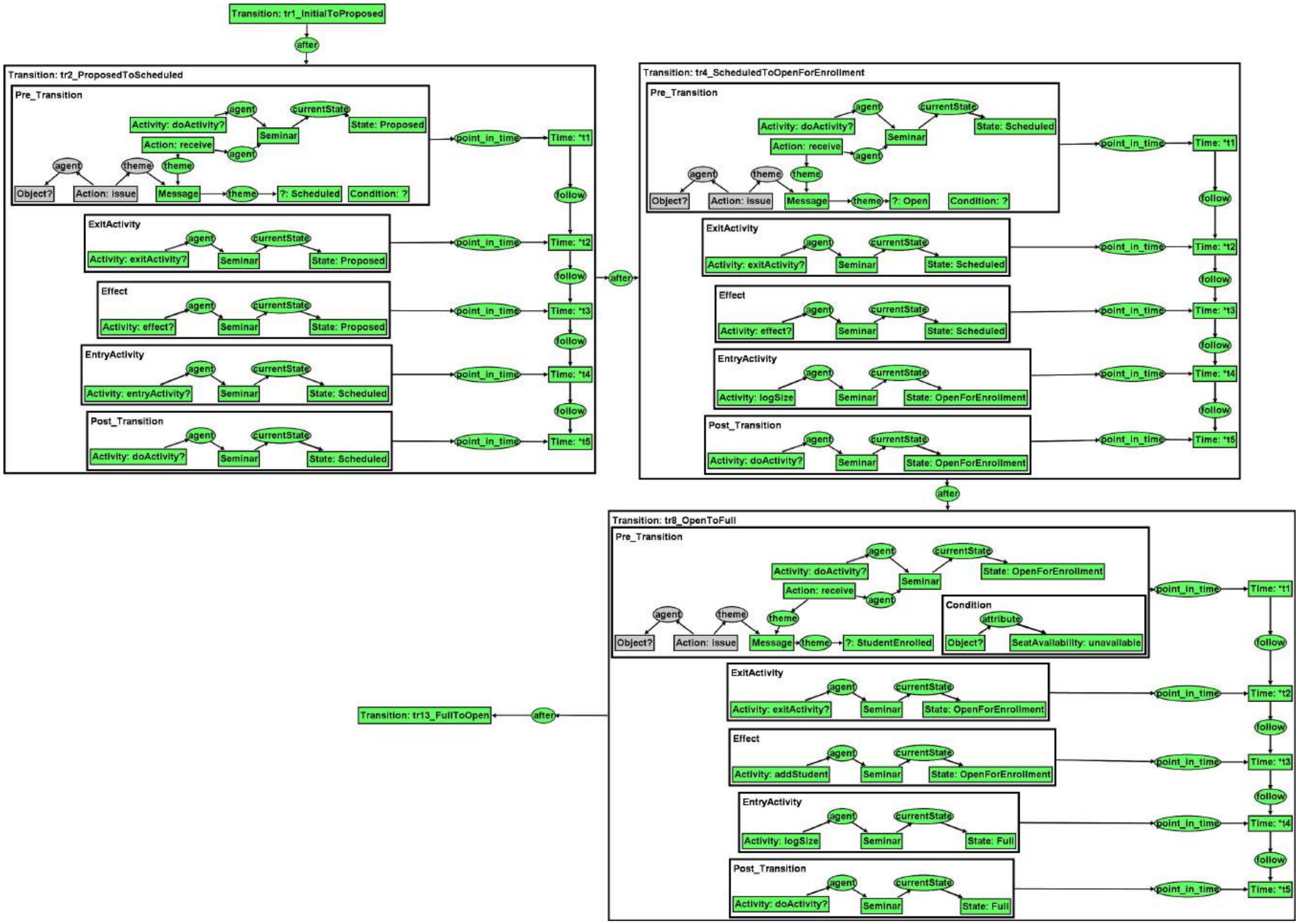


Figure A.35: UnivSys. generated sequence 2 in CGs

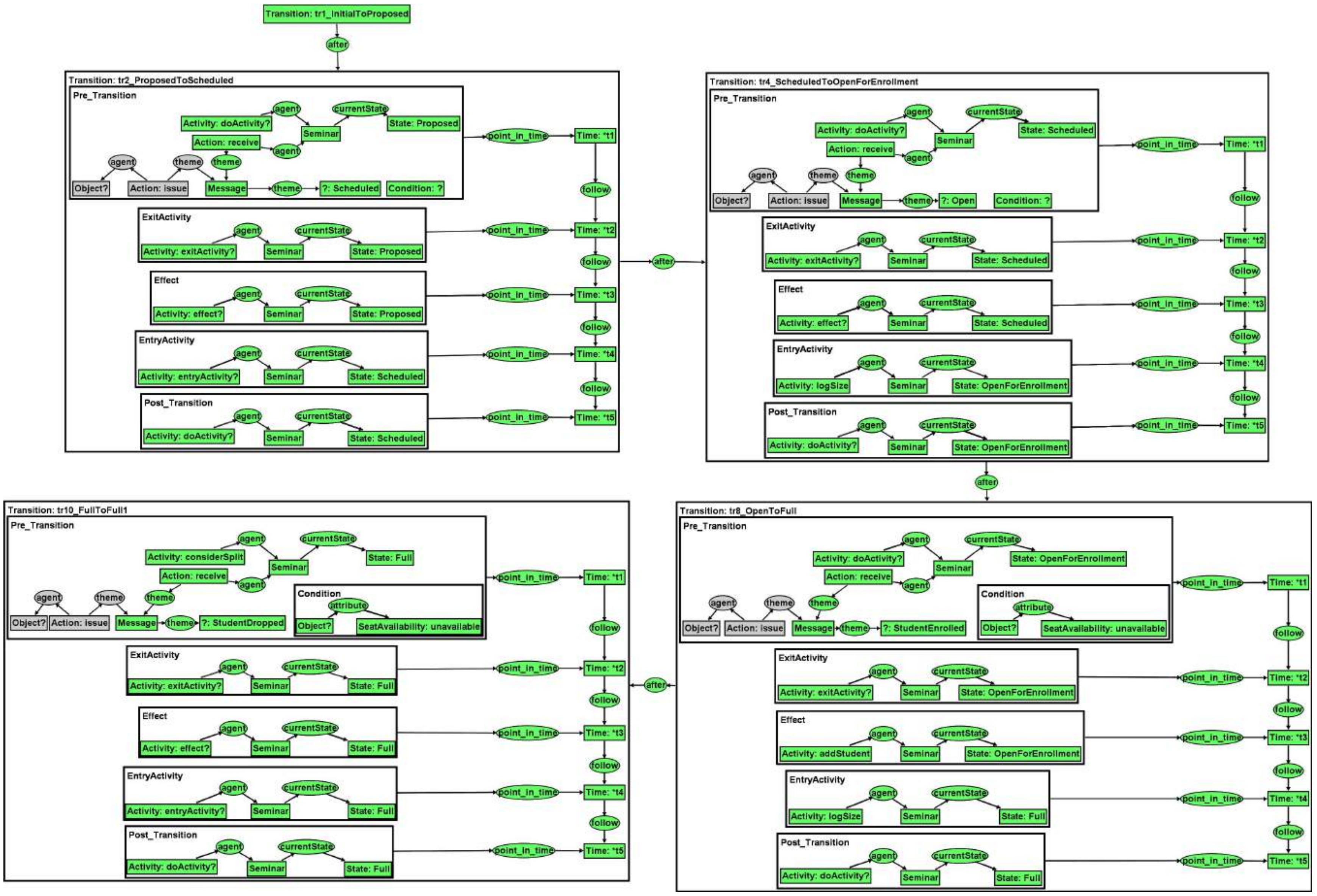


Figure A.36: UnivSys. generated sequence 3 in CGs

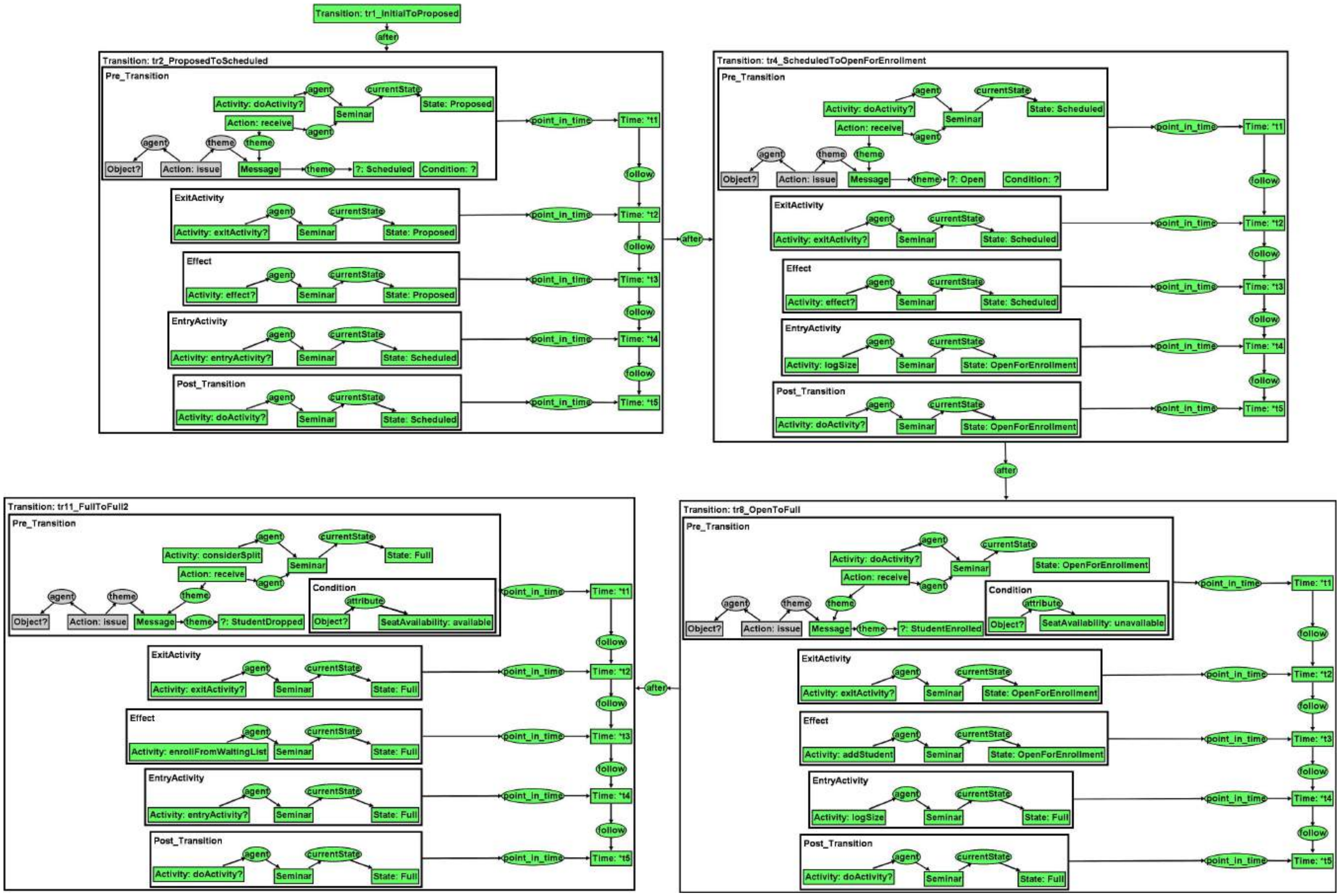


Figure A.37: UnivSys. generated sequence 4 in CGs

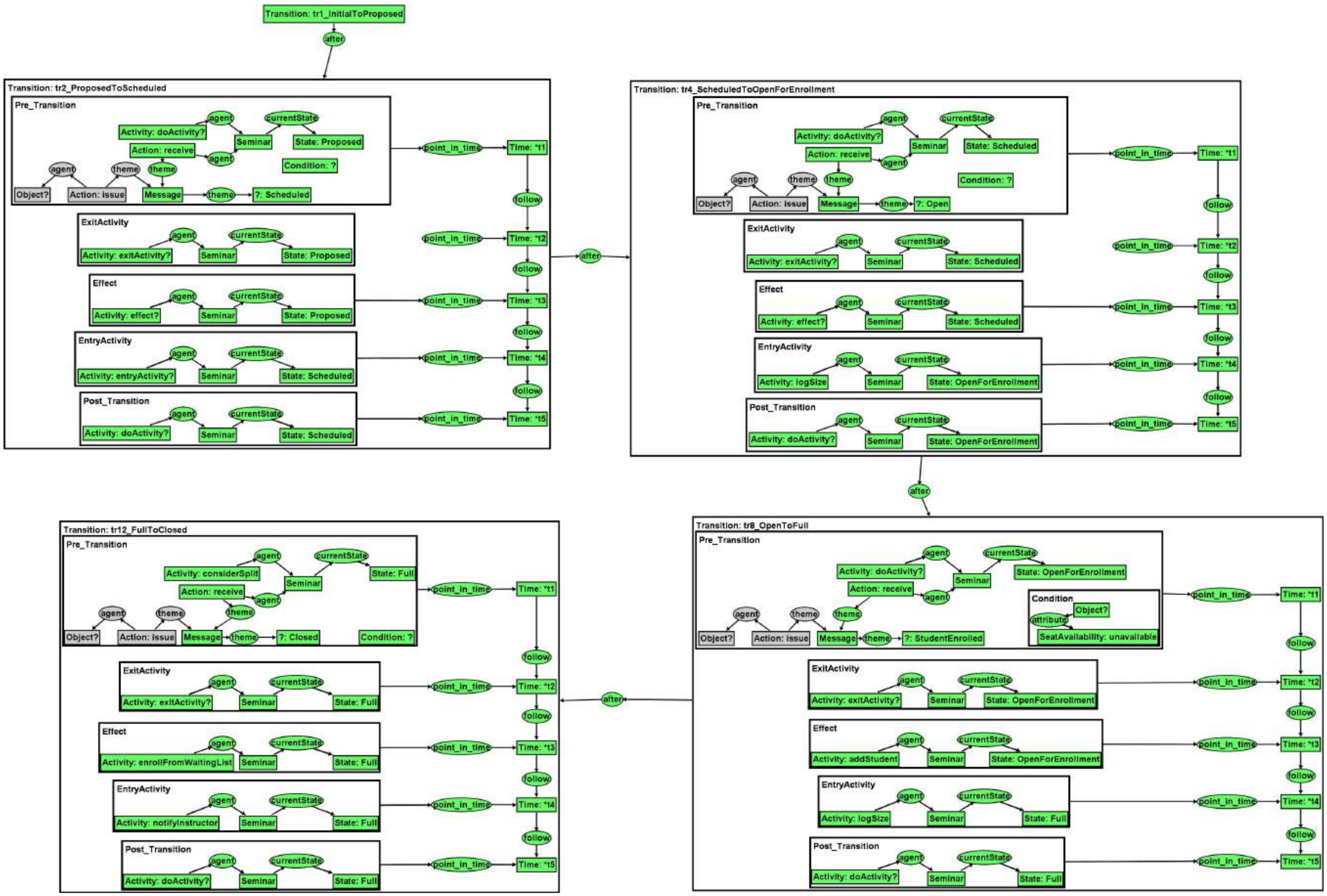


Figure A.38: UnivSys. generated sequence 5 in CGs

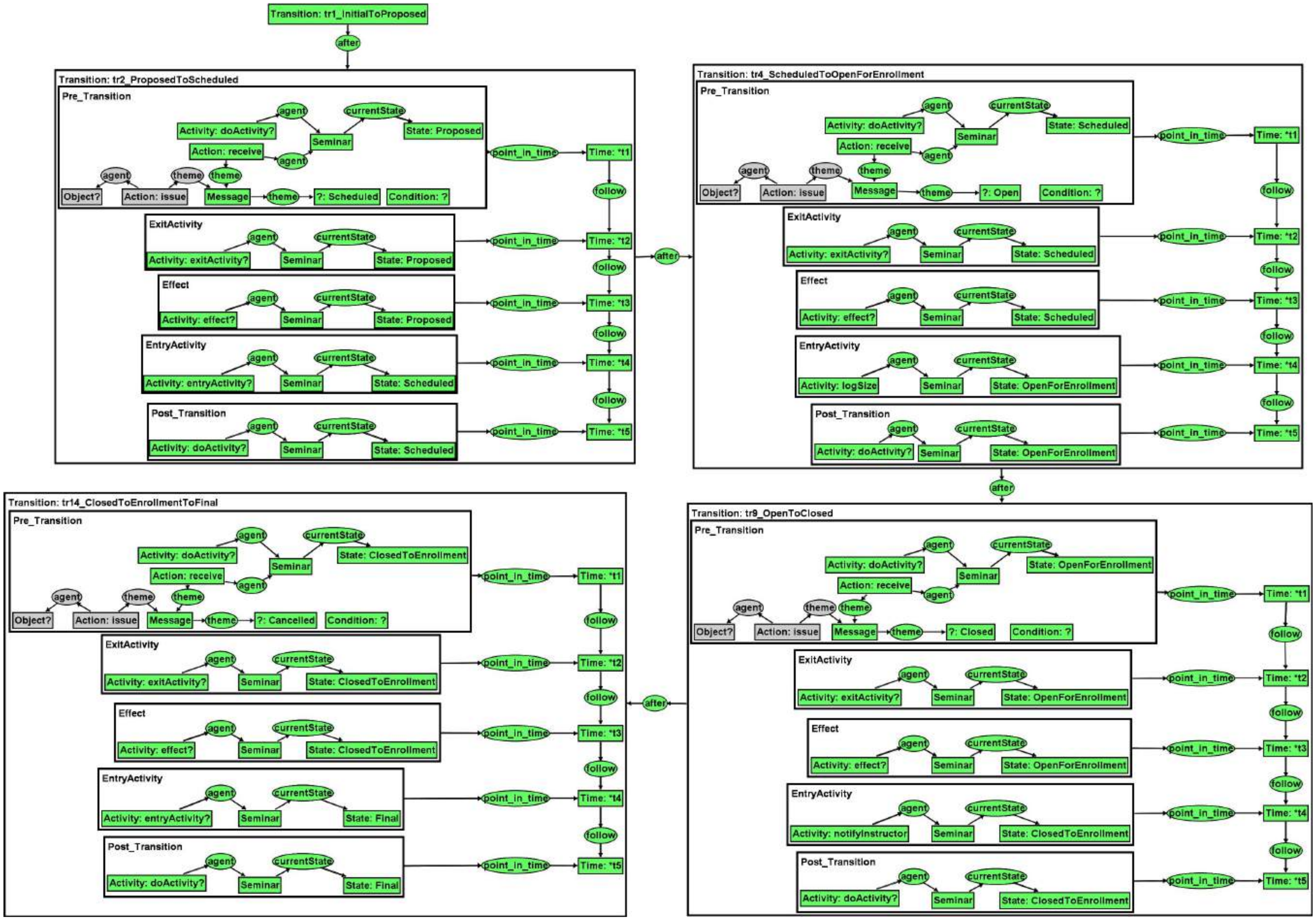


Figure A.39: UnivSys. generated sequence 6 in CGs

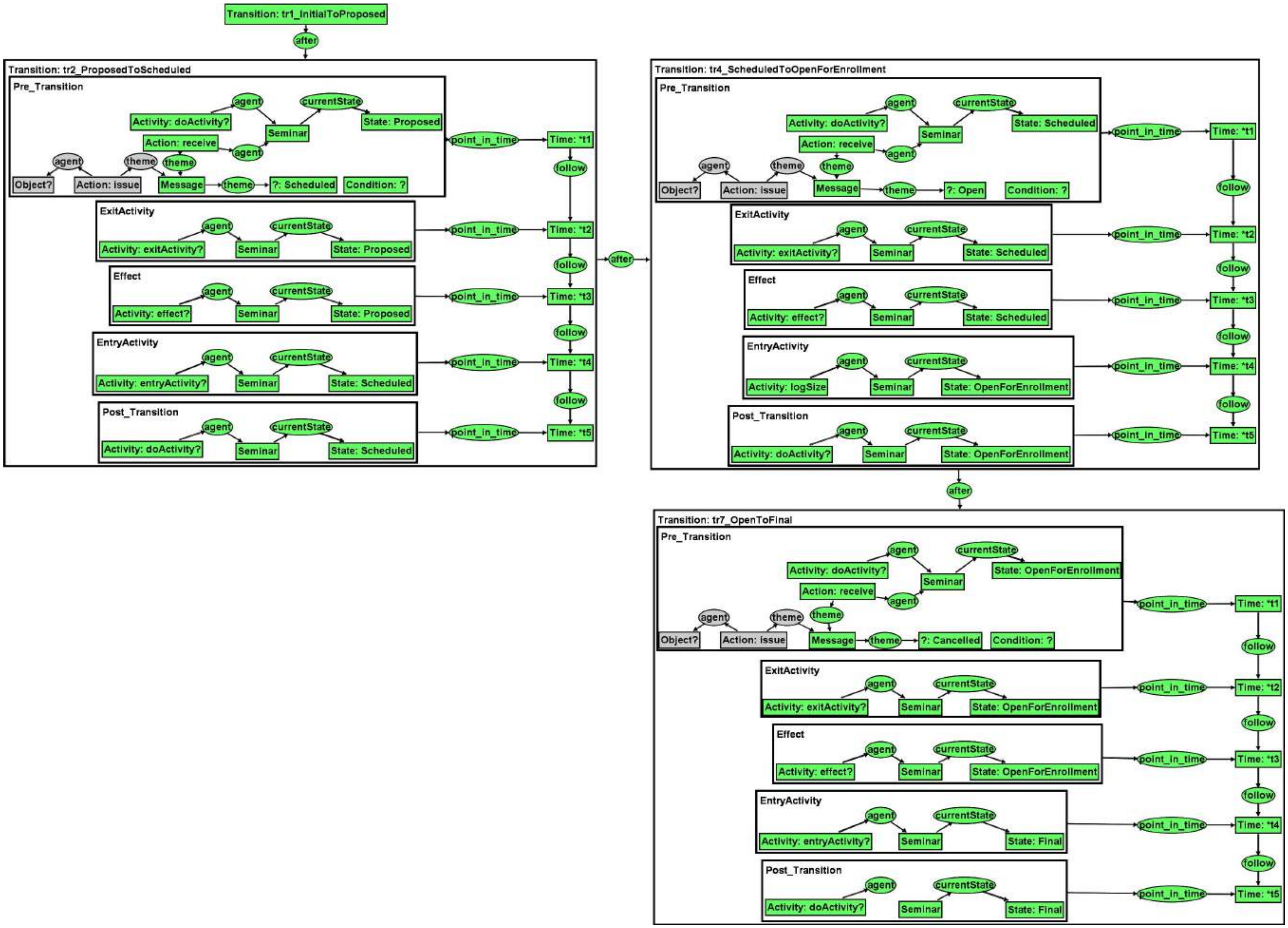


Figure A.40: UnivSys. generated sequence 7 in CGs

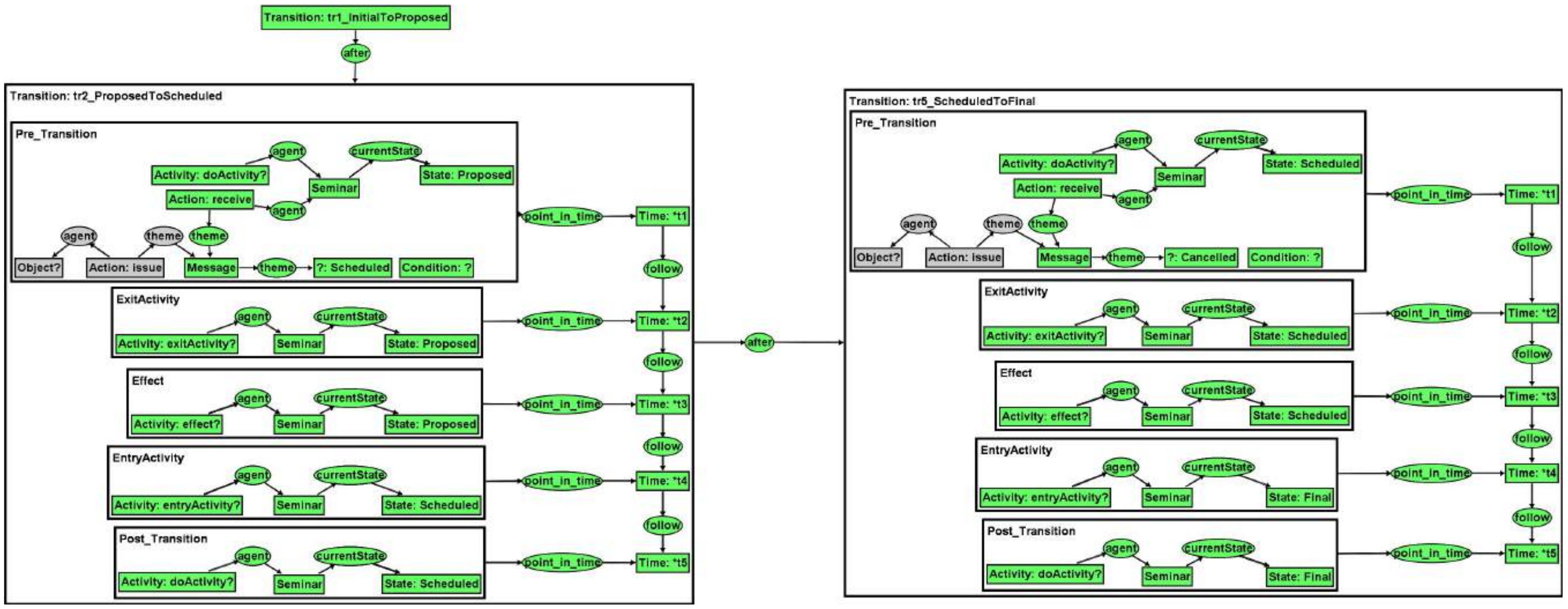


Figure A.41: UnivSys. generated sequence 8 in CGs

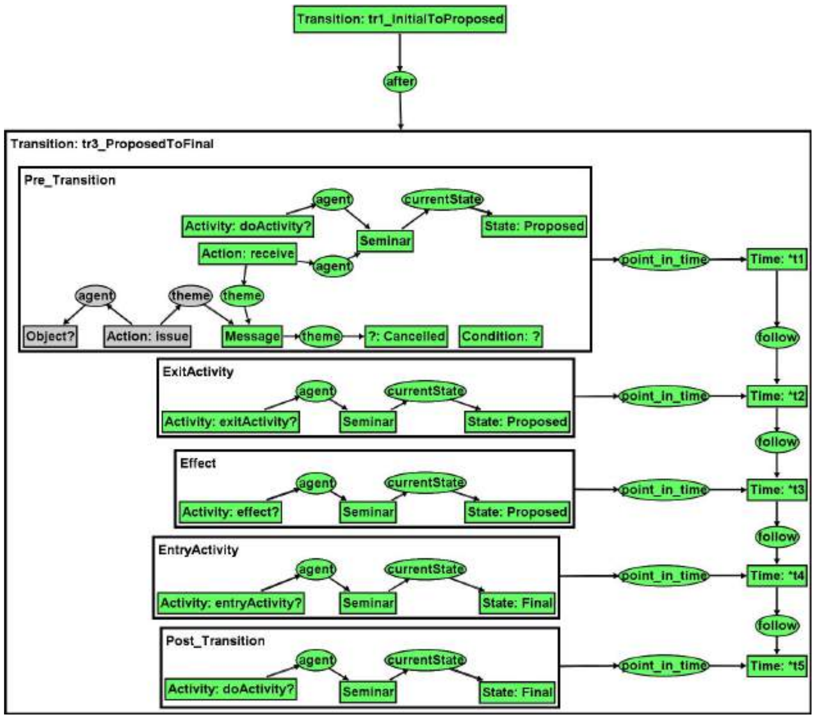


Figure A.42: UnivSys. generated sequence 9 in CGs

Then the nine sequences in CGs (Figure A.34 to Figure A.42) are translated back to the UML sequence diagram notations (Figure A.43 to Figure A.45). Note nine sequence diagrams are generated based on the nine sequences identified in the CGs Reservoir.

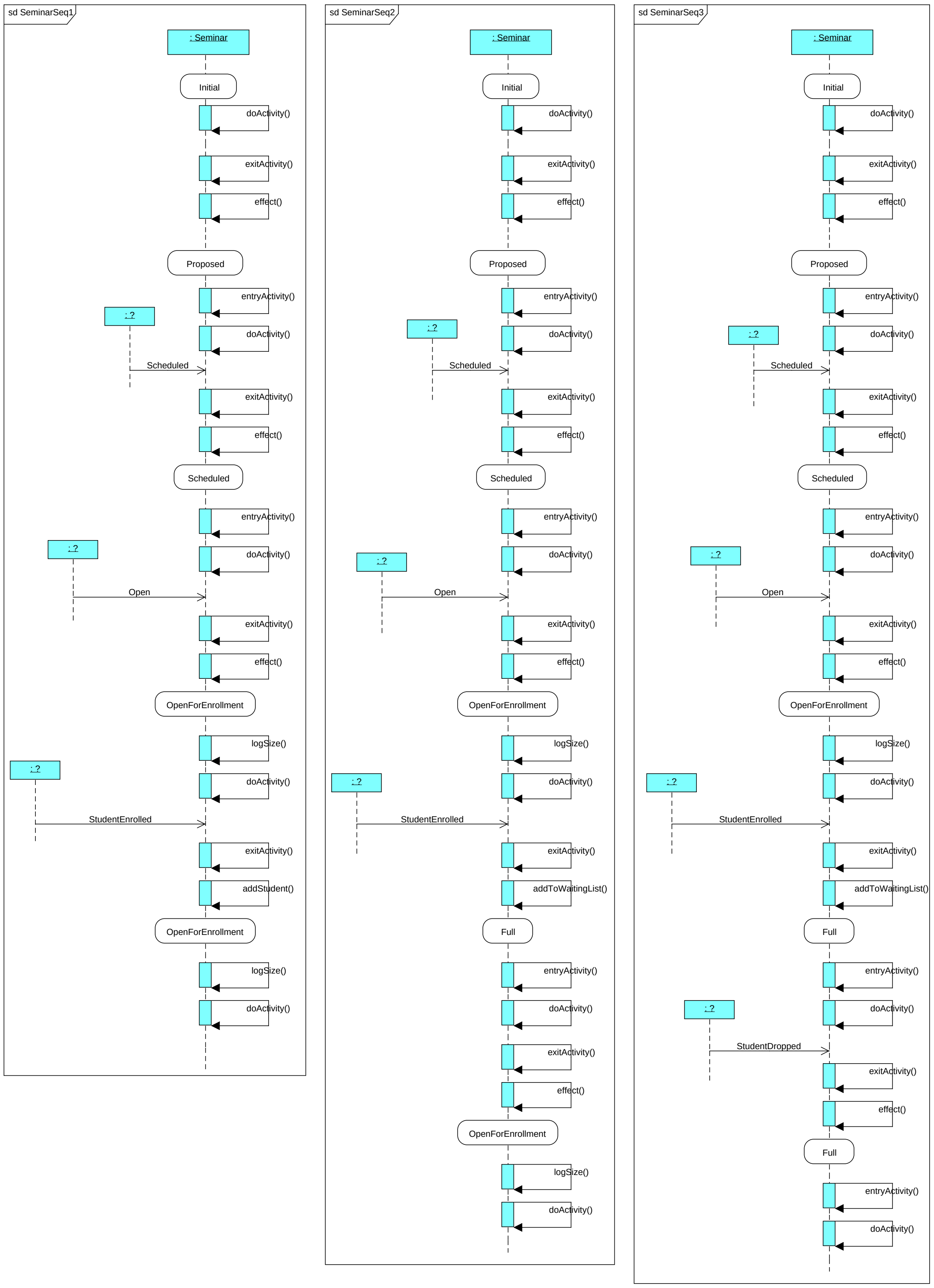


Figure A.43: UnivSys. generated sequence diagrams in CRF part 1

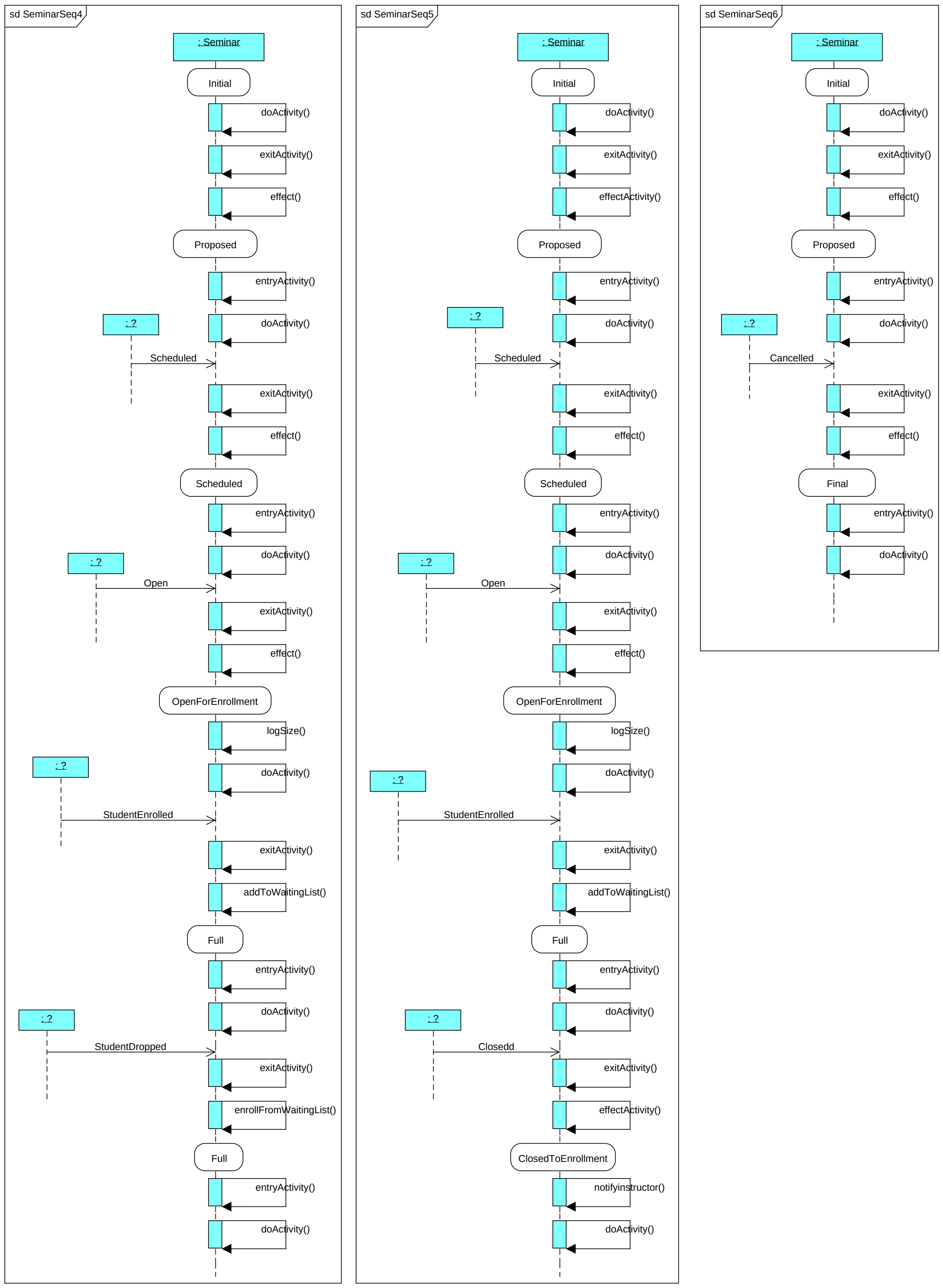


Figure A.44: UnivSys. generated sequence diagrams in CRF part 2

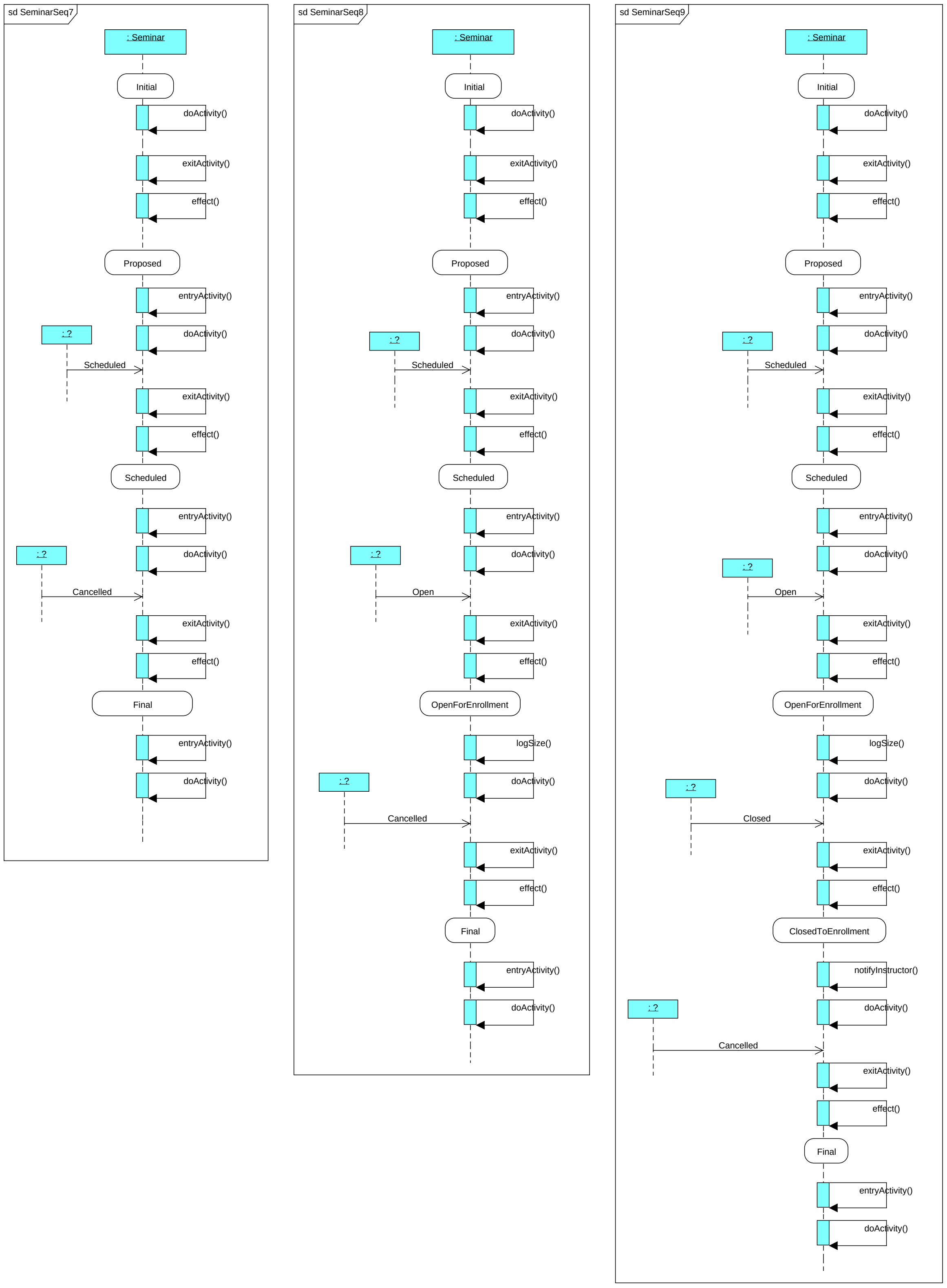


Figure A.45: UnivSys. generated sequence diagrams in CRF part 3

REFERENCES

[1] Donald Firesmith. Are your requirements complete? *Journal of Object Technology*, 4(1):27–44, 2005.

[2] Harry S Delugach. An approach to conceptual feedback in multiple viewed software requirements modeling. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints' 96) on SIGSOFT'96 workshops*, pages 242–246. ACM, 1996.

[3] Bashar Nuseibeh, Jeff Kramer, and Anthony Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *Software Engineering, IEEE Transactions on*, 20(10):760–773, 1994.

[4] Petri Selonen, Kai Koskimies, and Markku Sakkinen. Transformation between uml diagrams. *Journal of Database Management*, 14(3):37–55, 2003.

[5] W Lewis Johnson, Martin S Feather, and David R Harris. Representation and presentation of requirements knowledge. *Software Engineering, IEEE Transactions on*, 18(10):853–869, 1992.

[6] Walling R Cyre. Capture, integration, and analysis of digital system requirements with conceptual graphs. *Knowledge and Data Engineering, IEEE Transactions on*, 9(1):8–23, 1997.

[7] Alexander Franz Egyed. *Heterogeneous view integration and its automation*. PhD thesis, University of Southern California, 2000.

[8] Thanwadee Sunetnanta and Anthony Finkelstein. Automated consistency checking for multiperspective software specifications. In *Workshop on Advanced Separation of Concerns. Toronto*, 2001.

[9] Ragnhild Van Der Straeten, Tom Mens, Jocelyn Simmonds, and Viviane Jonckers. Using description logic to maintain consistency between uml models. In *«UML» 2003-The Unified Modeling Language. Modeling Languages and Applications*, pages 326–340. Springer, 2003.

[10] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. Specifying overlaps of heterogeneous models for global consistency checking. In *Proceedings of the First International Workshop on Model-Driven Interoperability*, pages 42–51. ACM, 2010.

[11] John F Sowa. Conceptual structures: information processing in mind and machine. 1983.

[12] Dong Liu, Kamalraj Subramaniam, Behrouz H Far, and Amin Eberlein. Automating transition from use-cases to class model. In *Electrical and Computer Engineering, 2003. IEEE CCECE 2003. Canadian Conference on*, volume 2, pages 831–834. IEEE, 2003.

[13] Boris Shishkov, Zhiwu Xie, K Lui, and J Dietz. Using norm analysis to derive use case from business processes. In *5th Workshop on Organizations semiotics. June*, pages 14–15, 2002.

[14] Kai Koskimies, Tarja Systä, Jyrki Tuomi, and Tatu Männistö. Automated support for modeling oo software. *Software, IEEE*, 15(1):87–94, 1998.

[15] Gregor Engels, Reiko Heckel, Gabriele Taentzer, and Hartmut Ehrig. A combined reference model-and view-based approach to system specification. *International Journal of Software Engineering and Knowledge Engineering*, 7(04):457–477, 1997.

[16] Thanwadee Thanitsukkarn and Anthony Finkelstein. A conceptual graph approach to support multiperspective development environments. In *Proceedings of the 11th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW'98)*. Citeseer, 1998.

[17] Li-jun SHAN and Hong ZHU. A formal descriptive semantics of uml. *Computer Engineering & Science*, 3:026, 2010.

[18] Francisco J Lucas, Fernando Molina, and Ambrosio Toval. A systematic review of uml model consistency management. *Information and Software Technology*, 51(12):1631–1645, 2009.

[19] Qvt Omg. Meta object facility (mof) 2.0 query/view/transformation specification. *Final Adopted Specification (November 2005)*, 2008.

[20] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, and Carolyn Talcott. Maude manual (version 2.6). *University of Illinois, Urbana-Champaign*, 1(3):4–6, 2011.

[21] Franz Baader. *The description logic handbook: theory, implementation, and applications*. Cambridge university press, 2003.

[22] Carlos Jaramillo, Alexander Gelbukh, and Fernando Isaza. Pre-conceptual schema: A conceptual-graph-like knowledge representation for requirements elicitation. *MICAI 2006: Advances in Artificial Intelligence*, pages 27–37, 2006.

[23] Ryo Hasegawa, Motohiro Kitamura, Haruhiko Kaiya, and Motoshi Saeki. Extracting conceptual graphs from japanese documents for software requirements modeling. In *Proceedings of the Sixth Asia-Pacific Conference on Conceptual Modeling-Volume 96*, pages 87–96. Australian Computer Society, Inc., 2009.

[24] George Spanoudakis. Analogical reuse of requirements specifications: A computational model. *Applied artificial intelligence*, 10(4):281–305, 1996.

[25] Anne Dardenne, Axel Van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of computer programming*, 20(1):3–50, 1993.

[26] Alan M Davis, Kathleen Jordan, and Tsuyoshi Nakajima. Elements underlying the specification of requirements. *Annals of Software Engineering*, 3(1):63–100, 1997.

[27] Don D Roberts. *The existential graphs of Charles S. Peirce*, volume 27. Walter de Gruyter, 1973.

[28] Lokendra Shastri. Semantic networks: An evidential formalization and its connectionist realization. 1987.

[29] Scott W Ambler. *The object primer: Agile model-driven development with UML 2.0*. Cambridge University Press, 2004.

[30] Grady Booch, Robert Maksimchuk, Michael Engle, Bobbi Young, Jim Conallen, and Kelli Houston. *Object-oriented Analysis and Design with Applications, Third Edition*. Addison-Wesley Professional, third edition, 2007.

[31] Axel Van Lamsweerde et al. Requirements engineering: from system goals to uml models to software specifications. 2009.

[32] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The*. Pearson Higher Education, 2004.

[33] Grady Booch. *The unified modeling language user guide*. Pearson Education India, 2005.

[34] Roy Grønmo and Birger Møller-Pedersen. From uml 2 sequence diagrams to state machines by graph transformation. *Journal of Object Technology*, 10(8):1–22, 2011.

[35] Bingyang Wei and Harry Delugach. *From State Diagrams to Sequence Diagrams: A Knowledge Acquisition Approach*, 2015.