INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

**ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600**

UMI[®]

UML ACTIVITY DIAGRAM REDUCTION BY GRAPH TRANSFORMATIONS

By

Ling He, B. Eng.

**A thesis submitted to the Faculty of Graduate Studies and Research in partial
fulfillment of the requirements for the degree of Master of Science in
Information and System Science**

**School of Computer Science
Carleton University
Ottawa, Ontario, Canada
September, 2001**

© Copyright 2001, Ling He

**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**395 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**395, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-66905-X

Canada

The undersigned recommend to the Faculty of Graduate Studies
and Research acceptance of the thesis

UML ACTIVITY DIAGRAM REDUCTION BY GRAPH TRANSFORMATIONS

Submitted by Ling He, B. Eng.

In partial fulfillment of the requirements for the degree of Master of Science
in Information and System Science

Frank Dehne, Chair
School of Computer Science

D. C. Petriu, Thesis Supervisor

Carleton University

September, 2001

Abstract

Performance analysis of a software design can give early warnings of performance problems, which can then be fixed easily and effectively at an early stage. Software Performance Engineering (SPE) is a technique that promotes the use of quantitative methods to analyze the performance of software systems during their development [Smith90]. This thesis contributes towards bridging the gap between software design and performance analysis. It proposes the use of UML activity diagrams annotated with performance information for realizing the first step of SPE. An activity diagram reduction algorithm is implemented by using a graph transformation tool named PROGRES (Programmed Graph Rewriting System). The proposed method is applied to two case study systems. This thesis is a part of a larger effort to translate UML models into Layered Queueing Network performance models.

To My Mother and Father

Acknowledgments

I would like to deeply thank my supervisor, Professor Dorina C. Petriu, for her encouragement throughout this research. Her consistent support and guidance have been crucial to the success of this work. Also, her inspiration and motivation will have a long-lasting influence on my future endeavors.

I thank my husband, Chen Mei, for his understanding, and moral support. My son, Henry, has always been my source of great joy and pleasure.

The financial assistance of the Communications and Information Technology Ontario (CITO) is gratefully acknowledged.

Last, but not least, thanks to my parents for their constant encouragement, from which all springs. I dedicate my thesis to them.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Contributions	3
1.3	Thesis Outline	4
2	Background Literature	6
2.1	Software Performance Engineering	6
2.1.1	SPE Approaches	6
2.1.2	SPE Benefits	9
2.1.3	The Steps of Performance Assessment	9
2.2	Performance Models	11
2.3	Unified Modeling Language	13
2.3.1	The UML Views and Diagrams	14
2.3.2	The UML Metamodel	15
2.3.3	Activity Diagrams.....	18
2.4	Performance Analysis of UML Modles	20
2.4.1	Performance Profile	21
2.4.2	Transformation of UML Models into Performance Models	23
2.5	Programmed Graph Rewriting System	24
2.5.1	Components of PROGRES	25
2.5.2	PROGRES Graph Schema	27
2.5.3	PROGES Graph Rewriting Rules	29
3	SPE Software Execution Model	33
3.1	Software Execution Model Notation.....	33
3.1.1	A Case Study	35
3.2	The Best Time Analysis Algorithm for the Execution Graph	41
3.3	Execution Time Calculation for the CAS Case Study	45
3.4	UML Solution for the Software Execution Model	49
3.4.1	UML Sequence Diagram for the Software Execution Model.....	49
3.4.2	UML Activity Diagram for the Software Execution Model	50

4	Activity Diagram Metamodel	52
4.1	Describing Behaviour in UML	52
4.2	Activity Diagram and Associated Metamodel Elements	55
4.2.1	Model Elements for Activity Diagrams	58
4.2.2	Execution Time Specification	63
4.3	Metaobject Diagrams	64
5	Activity Diagram Reduction By Graph Transformations	69
5.1	Graph Transformations	69
5.2	PROGRES Graph Schema	71
5.2.1	Mapping the UML Notation to PROGRES Schema Notation	71
5.2.2	PROGRES Schema Transformation from UML Metamodel	74
5.3	PRGORES Graph Transformation Rules	80
5.3.1	General Transformation Principles for Reduction Activity Diagram	80
5.3.2	Production Rules	82
5.3.2.1	Sequence Structure Production Rule	83
5.3.2.2	Branch and Merge Structure Production Rule	85
5.3.2.3	LoopBegin and LoopEnd Structure Production Rule	89
5.3.2.4	Fork and Join Structure Production Rule	91
5.3.3	Transaction for Activity Diagram Reduction	93
5.4	Control Algorithm	97
5.5	How long it takes to execute the reduction algorithm	99
6	Case Study	101
6.1	Computer Aided Design Case Study	101
6.1.1	Message Sequence Charts with Execution Graph	101
6.1.2	Using Activity Diagram to Analyze Performance	109
6.1.2.1	The Activity Diagram for Three Architectures	109
6.1.2.2	The Results for the Case Study	119
6.2	ATM Banking System Case Study	119
6.2.1	General Description of the Case	119
6.2.2	Scenarios Modeled by Activity Diagrams	121
6.2.3	Verification	129
6.3	Test Case for Multiple Branches	129

7	Conclusions	135
7.1	Conclusions	135
7.2	Future work	136
	References	138
Appendix A	Parts of the PROGRES Program for the ICAD Case Study	142
Appendix B	PROGRES code for Multiple Branches	144

List of Figures

Figure 2.1	Model Views and Diagrams	15
Figure 2.2	The UML Metamodel Architecture	18
Figure 2.3	Object Flows	20
Figure 2.4	The Performance Analysis Domain Model	22
Figure 2.5	Applying a Production Rule to a Host Graph	27
Figure 2.6	The Notation of PROGRES Graph Schema	28
Figure 2.7	An Example of a Test	29
Figure 2.8	An Example of a Production	30
Figure 2.9	An Example of a Transaction	31
Figure 2.10	An Example of a Function	31
Figure 2.11	An Example of the Import Type and its Functions	32
Figure 3.1	The Execution Graph Model of the DRAWMOD Scenario	37
Figure 3.2	The Execution Graph Models of the EDMS. DBC processing is in (a), ... the Find routine is in (b), the Retrieve routine is in (c), DB_MP Initiation is in (d), and DB_MP Termination is in (e)	38
Figure 3.3	The Execution Graph Models of the SYS_EXEC Communication Processing, the EDMS call processing is in (a), the SYS_EXEC loop is in (b), and the SYS_EXEC processing is expanded in (c).	41
Figure 3.4	Sequential-path Reduction Rule	42

Figure 3.5	Repetition-path Reduction Rule	42
Figure 3.6	Conditional-path Reduction Rule	43
Figure 3.7	Fork and Join Reduction Rule	43
Figure 3.8	The Basic Path Reduction Rules Produce the Total Computer Resource Requirements for the DRAWMOD Scenario	47
Figure 3.9	Activity Diagram Example.....	51
Figure 4.1	Possible Alternatives for Modeling the Sending/Receiving/of an Asynchronous	53
Figure 4.2	UML metamodel for Activity Diagrams (It Includes Elements from Action Graph, State Machine and Common Behaviour Diagrams)	57
Figure 4.3	Sequence Structure	64
Figure 4.4	Branch-Merge Structure	65
Figure 4.5	Fork-Join Structure	66
Figure 4.6	Loop Structure	67
Figure 5.1	Model Transformation by Using Graph Grammars	70
Figure 5.2(a)	UML Metamodel for Activity Diagrams	76
Figure 5.2(b)	PROGRES Schema for Activity Diagrams	77
Figure 5.3	Production Rule for the Reduction of a Sequence Structure	84
Figure 5.4	Production Rule for the Reduction of More Than Two Branches.....	86
Figure 5.5	Production Rule for the Reduction of the last Branch/Merge Structure	88
Figure 5.6	Production Rule for the Reduction of a Loop Structure	90
Figure 5.7	Production Rule for the Reduction of a Fork/Join Structure	92
Figure 5.8	Transaction Rule for the Reduction of All Sequence Path Nodes	94

Figure 5.9	The Reduction of a Graph with Several Sequence Structures	95
Figure 5.10	A Different Reduction Order for the Last Steps of Figure 5.9	96
Figure 5.11	The Algorithm for the Reduction of a Activity Diagram	97
Figure 6.1	The DrawModel Scenario	102
Figure 6.2	The DrawModel Use Case Scenario for Architecture 1	105
Figure 6.3	The DrawModel Use Case Scenario for Architecture 2	106
Figure 6.4	The DrawModel Use Case Scenario for Architecture 3	107
Figure 6.5	Activity Diagram for Architecture 1 of Draw Model Scenario	111
Figure 6.6	Activity Diagram for Architecture 2 of Draw Model Scenario	114
Figure 6.7	Activity Diagram for Architecture 3 of Draw Model Scenario	117
Figure 6.8	Use Case Model for Banking System	121
Figure 6.9	High-Level Activity Diagram for ATM Banking System Case Study	123
Figure 6.10	Activity Diagram for Validate PIN in ATM Case Study	124
Figure 6.11	Activity Diagram for Withdraw Funds in ATM Case Study	125
Figure 6.12	Activity Diagram for Query Account in ATM Case Study	127
Figure 6.13	Activity Diagram for Transfer Funds in ATM Case Study	128
Figure 6.14	Six Branches Structure and Combination other Structures	131
Figure 6.15	Input Graph for a Test Case	132
Figure 6.16	The Final Graph and Execution Time for the Case	133

List of Tables

Table 3.1	Execution Graph Notation Summary	34
Table 5.1	UML Metamodel vs. PROGRES Schema Notation	72
Table 6.1	The Response Time for Each Architectures by SPE.ED Tool	108
Table 6.2	The Response Time for Each Architectures By PROGRES Tool	119
Table 6.3	The Execution Time for ATM Banking System Case Study	129

1.2 Motivation

The performance of a software system is crucial to its use, especially in the case of real-time and distributed systems. As software systems become larger, more complex, and include more features, it becomes essential to be able to predict performance problems well into the early phases of the software lifecycle, when the problems can be fixed more easily. The early evaluation of performance characteristics was proposed by C. Smith who introduced the idea of Software Performance Engineering (SPE) in [Smith90].

SPE is a technique that proposes the use of quantitative methods and performance models in order to assess the performance effects of different design and implementation alternatives during the development of a system [Smith 90]. SPE analysis can identify, solve, and prevent performance problems by suggesting corrective measures in the early development stages, when the corrections are easier to apply. SPE may utilize a combination of experience, performance measurements, and performance models. Performance models allow for the quantitative assessment of performance characteristics for software systems at an early stage, when the entire system is not yet implemented, and cannot, therefore, be measured. SPE contains two big steps. In the first step, a number of representative scenarios are selected and a so-called “software execution

model” is built for each scenario. A “software execution model” is a kind of a flow graph (see more details in chapter 3). Quantitative resource demands are attached to each flow graph block. By reducing the flow graph, aggregate resource demands are computed for every scenario. These can be used for obtaining performance characteristics (such as response time and throughput) in *the absence of contention* from multiple users. In the second SPE step, the software execution model is mapped onto a performance model (such as a Queuing Network) and the performance of the whole system is analyzed in *the presence of contention* from multiple users.

Recently, the SPE technique has been increasingly used to construct software systems to meet performance objectives. However, a gap still remains between the software development and the performance analysis domains. The gap is due to the lack of some simple, inexpensive, and scalable techniques and tools for analyzing the performance characteristics of design specification, such as those represented by UML models [UML1.4][Booch+99][Bochamnn00][Rumbaugh+99][Oestereich99][Pooley+99]. Clearly, some sort of automation is needed to transform UML models into performance models. This need was recognized by the Object Management Group (OMG) which is working on defining a performance profile [OMG01]. The profile extends the UML model with performance-related information, with the purpose of enabling quantitative analysis of UML models.

This thesis contributes to the effort of conducting performance analysis of UML models. We expressed the first step of SPE in the UML domain by using activity

diagrams instead of flow graphs. For expressing quantitative resource demands, we used attributes from the proposed OMG performance profile [OMG01].

The process of activity diagram reduction was implemented by using a graph rewriting tool, based on graph grammars, named PROGRES [Schüerr97a] [Schüerr97b] [Schüerr97c] [Schüerr94]. The graph rewriting (or graph transformation) system PROGRES contains a graph-oriented data model (or schema), a set of transformation rules for changing (i.e., rewriting) the graph, and an algorithm for applying the rules. In order to implement the reduction of activity diagrams with the PROGRES, we had to express the subset of the UML metamodel that describes activity diagrams as a PROGRES schema. Then, we had to define appropriate graph transformation rules and the algorithm for applying them (see Chapter 5 for more details).

Graph rewriting is not the only possible approach for implementing the reduction of activity diagrams. We have chosen this technique in order to align our work with a larger effort that aims at translating UML models into performance models by graph rewriting [Petriu01] [Petriu00a] [Petriu00b] [Petriu99] [Petriu98] [Wang99][Hode01]. The reason this work uses graph rewriting techniques is that both the “input model” (i.e., a UML model) and the “output model” (i.e., a performance model) can be expressed as graphs. Graph rewriting (a.k.a. graph transformation) techniques are very appropriate for translating an input graph of a certain kind into an output graph of another kind.

1.3 Contributions

The main contributions of this research are as follows:

- Expressed the first step of SPE in the UML domain, by using activity diagrams instead of flow graphs. We also used elements of the new performance profile proposal [OMG01] to express quantitative resource demands.
- Implemented the activity diagram reduction by using an existing graph transformation tool PROGRES. We have translated the UML metamodel subset for activity diagrams into a PROGRES schema and defined appropriate transformation rules and algorithm. The PROGRES program was tested on a number of test cases.
- Applied the proposed SPE step based on activity diagrams to two case studies: the CAD and the ATM banking system.

1.4 Thesis Outline

The thesis is organized as follows:

Chapter 2 briefly presents an overview of the background information related to this thesis, such as Software Performance Engineering (SPE), Performance Models, Unified Modeling Language (UML), UML and Performance Analysis, and Programmed Graph Rewriting System (PROGRES).

Chapter 3 presents in detail the first step of SPE analysis as introduced in [Smith90] and discusses how UML can be used instead.

Chapter 4 presents the details of using activity diagrams instead of Smith's flow graphs. The UML metamodel related to an activity diagram is discussed, and examples of metaobject diagrams are given for different diagram structures.

Chapter 5 describes the PROGRES program used to reduce activity diagrams according to the first SPE step. The graph schema, graph transformation rules, and a control algorithm are presented in detail.

Chapter 6 discusses two client/server case studies named CAD and ATM Banking System, and a multiple branches testing case. These case studies are implemented to demonstrate and verify the proposed transformation process.

Chapter 7 contains the main conclusions of the thesis, summarizes the contributions, and describes suggestions for future work.

Chapter 2 Background Literature

This chapter presents an overview of the background information related to the thesis. It contains a brief review of five main areas: Software Performance Engineering (SPE), Performance models, the Unified Modeling Language (UML), UML to Performance Analysis, and Programmed Graph Rewriting System (PROGRES).

2.1 Software Performance Engineering

Performance characteristics (such as response time and throughput) play an important role in defining the quality of software products, especially in the case of real-time and distributed systems. The attitude that performance problem can be fixed later has proven to be very costly and inefficient. A better approach is to use Software Performance Engineering (SPE) to ensure that the performance expectations of a software system are met as it is being designed and implemented. SPE is a method for constructing software systems to meet their performance objectives [Smith90] [Smith+93] [Smith+97] [Smith+98] [Williams+95] [Williams+98].

2.1.1 SPE Approaches

SPE is used to enhance performance or address performance issues throughout the lifetime of a software system. The term “performance” refers to throughput, response time, and utilization as performance characteristics seen by the end user of a software system. SPE may use different techniques to analyze performance.

In general, performance analysis can be done at different levels. A first level of analysis may use experience or intuition. This approach may be appropriate if something similar to the proposed system already exists and an “expert” is available. However, it is difficult to apply to large systems because of their complexity.

Another analysis approach is to measure performance metrics which are obtained through experiments or normal use of a system [Fenton97]. Measurements are considered to be trustworthy since they are taken from the final (or a similar) system. However, even with a small system it is difficult to isolate the system variables and application behavior to allow for the generalization of the measurement data [Jain91]. Extrapolation of measurements, such as estimating how software will scale, is difficult. This approach may also be impractical due to the system size, expense, or lack of availability. In particular, metrics of a distributed system are difficult to obtain, because the control and data-flow is spread among computing nodes. In the case of SPE applied to systems that are designed but not entirely implemented, the whole system cannot yet be measured. It is possible to use measurements from other similar systems, and estimations or time budgets for the parts not yet implemented [Smith90].

Another approach to performance analysis is by constructing and analyzing performance models. A model should emphasize key performance aspects of a system by combining behavioral and structural aspects of workload elements in terms of resource usage [Jain91]. There are two main classes of performance models for software and computer systems: simulation and analytical models. While a simulation model may capture more details of the system under study than analytical models, it has the

disadvantage that it takes longer to evaluate. On the other hand, the analytical models, while they are more efficient at evaluation than the simulation models, are more restrictive, in the sense that they may be able to represent fewer details of the real system. Two well known types of analytical models are Queueing Networks[Lazowska+84] and Petri Nets [Chichester95]. A performance model is evaluated to produce performance predictions that are compared with the performance requirements (e.g., utilization or delay values). Performance models are flexible, providing predictions for varying environmental conditions or design alternatives. So, if the performance requirements are not met, the predictions can help identify the source of the problem and the model can be used to explore potential corrections. If a new system is being analyzed, modeling is the only analysis approach available [Lavenberg89]. SPE uses performance models that may be very rough and simplified in the first development stages. As the development evolves, the performance models may capture more details and become more accurate.

Connie Smith proposed a systematic way to use quantitative methods and models to assess requirements, design and hardware alternatives, starting early in the software life-cycle when a wide range of options exists, and continuing through the life-cycle [Smith90]. The method derives a model of a software execution graph from the design, constructs a second model for performance predictions, and then uses the predictions to guide the modification of the design.

Performance modeling can be used with other analysis approaches. A model can identify the key parameters to measure. If measurements do detect a problem, the model

can help designers to understand the problem. Measurements can also provide feedback for a documented model by validating its predictions.

2.1.2 SPE Benefits

SPE can achieve the performance goals with little or no additional development time or cost without sacrificing understandability and maintainability. The benefits of SPE are as follows [Smith90]:

- *Increased productivity*, because optimization is performed early at high levels of abstraction rather than later at the code-tuning level.
- *Improved quality and usefulness of the resulting software product* by selecting suitable design and implementation alternatives, thus avoiding widespread tuning modifications.
- *Controlled costs of the supporting hardware and software* by identifying what equipment is needed and allowing sufficient time for competitive procurement.
- *Enhanced productivity during the implementation, testing, and early operational stages* by ensuring that sufficient computing power is available
- *Improved user relations because it is customized* to fit its most frequent uses by avoiding long interim periods of poor responsiveness.

2.1.3 The Steps of Performance Assessment

Using a model for performance analysis consists of two main steps: the model generation and the model evaluation. For every development phase, SPE builds two models, as shown in [Smith90]:

I. Build and analyze the Software Execution Model

- Build the Software Execution Model, a flow-graph that follows the execution of the software.
- Gather resource requirements for every software module, then aggregate the resource requirements in order to assess the performance in the “no contention” case.

II. Build and analyze the System Model

- Map the Software Execution Model to a System Model (usually a Queueing Network) that contains both the hardware and the software of the system.
- Solve the model analytically or with simulation to analyze the performance in the presence of contention from multiple users. Both best and worst case analysis may be conducted.
 - If the best-case performance is unsatisfactory, seek feasible alternatives.
 - If the worst-case performance is satisfactory, go into the software development cycle.

Performance modeling, provides a basis for gathering, organizing, evaluating, and understanding information about a computer system [Lazowska+84]. SPE aims to insure that the system will meet its performance objectives by the means of assessing the performance effects of different design and implementation alternatives during the development of a system. The need for SPE is generally recognized by industry, but is not widely applied.

One of the most difficult steps in SPE is to gather quantitative data for resource usage estimation, such as execution times. It is seldom possible to obtain precise information for all the activities performed in a scenario. In early design phases, performance analysis is based on bounded estimates for resource demands, educated guesses based on previous experience, or time budgets for different operations. Also, if some components are reused, these can be measured and “real” values can be used for SPE. As the project progresses, more accurate numbers may become available, so the performance models will give more precise results. In [Smith90], it is emphasized that it is better to start early with an imprecise estimation than to wait too long until a precise number can be obtained. In later phases it is much harder to make the radical changes that may sometimes be necessary in order to meet the performance requirements.

2.2 Performance Models

There are different types of performance models. Among the best known are the Queueing Network model (QN)[Lazowska+84] and the Petri Nets model [German+00] [Chichester95]. SPE uses mostly QN models. A QN model is a collection of service centers, each with its own queue, and customers or jobs, which represent the consumers of service. Each service center represents a common resource shared by customers, and has an established service discipline. After the completion of service at one service center, a customer may move to another service center according to a routing chain. Except for contention for common resources, the customers are independent. The model usually represents hardware resources as pure servers and software tasks as pure clients. It is hard to capture complex interactions among various software and hardware

components in client-sever distributed processing systems. The QN model cannot easily capture the interactions in client/server systems with nested servers. An extension of QN named Layered Queueing Network (LQN) was developed for handling such complex interactions. Layered Queueing models were independently developed as Stochastic Rendezvous Networks (SRVN or SRN) in [Woodside89] [Petriu91b] [Petriu94] [Woodside95a] [Woodside98] and as the Method of Layers (MOL) in [Rolia87] [Rolia92] [Rolia95]. Many features of the two approaches were joined under the name of LQN [Frank95] [Woodside95b].

LQN model is capable of modeling most of the features which are important for performance, such as multi-threaded processes, devices, locks, communications, and so on. It represents software resources in a natural way so that they are part of the framework, and thus special approximations do not have to be developed for every system [Woodside+95]. The model is closely linked to software descriptions and provides a transparent representation of the software architecture, which makes models easy to develop and understand. The model is well suited for systems with parallel processes running on a multiprocessor or on a network, such as a client-server system.

The effort to transform UML models into performance models, of which this thesis is only a part, aims to generate LQN models [Petriu01] [Petriu00a] [Petriu+00b] [Petriu+99] [Petriu+98] [Wang99][Hoda01].

2.3 Unified Modeling Language

The Unified Modeling language (UML) is a graphical language and notation for specification, construction, visualization, and documentation of models of software systems. UML has become the standard method of writing a system's blueprints in the software industry [Booch+98].

UML was developed under the direction of Grady Booch, Ivar Jacobson, and James Rumbaugh of Rational Software [Booch+98] [Rumbaugh+99] [Jacobson+99]. Many other enterprises have actively participated in its development and currently support UML [UML1.3][Bezivin+98][Douglass98][Eriksson+98][Fowler+99].

According to [UML1.3], the primary design goals of the UML are as follows:

- Provide users with a ready-to-use, expressive visual modeling language to develop and exchange meaningful models.
- Have extensibility and specialization mechanisms to extend the core concepts.
- Support specifications that are independent of particular programming languages and development processes.
- Provide a formal basis for understanding the modeling language.
- Encourage the growth of the object tools market.
- Support higher-level development concepts such as components, collaborations, frameworks and patterns.
- Be scalable and widely applicable (general purpose and powerful) and usable (simple, widely accepted, evolutionary)
- Integrate best engineering practices.

According to [Douglass98], the main UML features are the following: object model, use case and scenarios, behavioral modeling with state charts, packaging of various kinds of entities, representation of tasking and task synchronization, models of physical topology, models of source code organization, and support for object-oriented pattern.

2.3.1 The UML Views and Diagrams

The UML supports the following five model views [Sinan98] to facilitate the organization of knowledge within problem solving as shown in Figure 2.1:

- *User model view* encompasses a problem and solution from the perspective of those individuals whose problem the solution addresses. This view presents the goals and objectives of the problem owners and their requirements of the solution.
- *Structural model view* encompasses the static, or structural, aspects of a problem and solution.
- *Behavioral model view* encompasses the dynamic, or behavioral, aspects of a problem and solution, also interactions or collaborations among problem and solution elements.
- *Implementation model view* encompasses the structural and behavioral aspects of the solution's realization.
- *Environment model view* encompasses the structural and behavioral aspects of the domain in which a solution must be realized.

The UML defines the following nine graphical diagrams [UML1.3]:

- Use Case Diagram

- Class Diagram
- Object Diagram
- Behavior Diagrams:
 - Sequence Diagram
 - Collaboration Diagram
 - Statechart Diagram
 - Activity Diagram
- Implementation Diagrams:
 - Component Diagram
 - Deployment Diagram

Figure 2.1 Model View and Diagrams

2.3.2 The UML Metamodel

The UML [OMG99] itself is defined by following a metamodeling approach. The metamodel encompasses the most important concepts from the most important object analysis and design, and related methods.

The UML metamodel is defined as one of the layers of a four-layer metamodeling architecture [UML1.4]. In this architecture the UML metamodel is an instance of the meta-metamodel of the OMG Meta-Object Facility (MOF), whose basic responsibility is to define a language for specifying model elements, such as Class, Attribute, and Operation [OMG].

The UML metamodel organizes metamodel elements within logical packages. Figure2.2 shows the structure of the UML metamodel, which is organized as a set of three UML packages containing classes. Each package describes a different aspect of the language, namely types, expressions, and values. Dashed arrows between two packages or model elements, where the source of the dashed arrow is dependent on the presence or existence of the target of the dashed arrow, indicate that a relationship exists between the elements. The UML is defined within the following packages [UML 1.4].

- *Model Management package* shows how modeling elements are organized. It defines packages as grouping units for other model elements.
- *Foundation package* defines the concepts required for structural modeling and the mechanisms for extending the language. The Foundation package is composed of the following packages:
 - *Data Types package* defines different data types used within the UML metamodel.
 - *Core package* defines the most fundamental concepts of the UML. Core concepts include classes and associations.
 - *Auxiliary elements package* defines other fundamental concepts of the UML that

extend the Core package.

- *Extension Mechanisms package* defines the concepts required to customize and extend the UML. These concepts are stereotypes, tagged values, and constraints.
- *Behavioral Elements package* defines the concepts required to model a system's behavior. It is composed of the following packages:
 - *Common Behavior package* defines the core concepts required for dynamic elements, which are for use cases, collaborations, and state machines.
 - *Use Cases package* defines the modeling of an entity's functions, or what it offers to outside users. These concepts include use cases and actors.
 - *Collaborations package* defines how different elements of a model interact to realize use cases. These concepts include collaborations and interactions.
 - *State Machines package* defines how control and data flow among elements of a model interact to realize use cases, which include events, states, and transitions.

The UML metamodel elements are divided into several class diagrams. Different diagrams are used for different purposes depending on the angle from which the system is viewed.

Figure 2.2 The UML Metamodel Architecture

2.3.3 Activity Diagrams

From Figure 2.1, we know that activity diagrams lie within the behavioral view of a system and describe the activities or actions of the classes participating in the system behavior. These diagrams describe the behaviors of a class in response to internal processing, rather than to external events as in statechart diagrams. Activity diagrams

contain action states, action flows, and object flows. They may be constructed using the following technique: identify the classes participating in behavior whose processing activities are to be specified, then model the swimlanes, the action states, the action flows, and the object flows [Sinan98].

Activity diagrams are variations of state machines or statechart diagrams, in which states are activities representing operations, and transitions are triggered by the completion of actions within states. Whereas a state machine describes the state of an object, an activity diagram may be associated with classes, methods, or use cases. Activity diagram may describe activities among a set of objects. The activity diagrams may have various expressions (guard conditions and actions) expressed using pseudocode or another language. They are types of interaction diagrams that focus on the flow of control and information within behavior, which may be used to validate use cases or for business modeling [Fowler+99].

Swimlanes

Swimlanes are graphical constructs that represent a partitioned set of actions. They are used to model the responsibilities of one or more objects for actions within an overall activity; that is, they divide the activity states into groups and assign these groups to objects that must perform the activities [UML 1.4].

Action States

Action states are states representing the execution of atomic actions or operations. They are used to model atomic actions of entities or steps in the execution of an algorithm [UML 1.4].

Action Flows

Action flows are associations between action states. They are used to model relationships between the action states of an entity. Action flows are activities performed by objects on objects. An entity in one action state will perform the specified action and enter another action state when the first action is completed [UML 1.4].

Object Flows

Object flows are associations between action states and objects. They are used to model the utilization of objects by action states, and the influence of action states on objects, as shown Figure 2.3.

Figure 2.3 Object Flows

Object flows are denoted as dashed paths, and specify the flow of objects between action states, which have object symbols [UML 1.4].

2.4 Performance Analysis of UML Models

The development of software systems which satisfy performance requirements is critical for using the software system. Although UML has become a standard notation for

software and hardware system modeling, it is still missing information for performance analysis. A performance profile will play a key role in extending UML.

2.4.1 Performance Profile

OMG is in the process of defining a number of profiles that extend UML for different application domains. Among them is the performance profile proposed in [OMG01]. The profile is intended to enable quantitative performance analysis of UML models, and to extend the UML metamodel for attaching performance information to a UML model with the following elements:

- Stereotypes
- Tagged values
- Constraints

In order to conduct quantitative performance analysis for UML model, several steps are necessary, as follows:

- One must first translate the UML model into a performance model
- Use an existing performance analysis tool to analyze the performance model for a range of alternatives and workloads
- Import the analysis results back in the UML model

Figure 2.4 shows the domain model related to the proposed performance profile [OMG01]. The performance analysis domain model depicts a general performance model that identifies the basic abstractions and relationships used in performance analysis. The concepts in this model are fully consistent with the conceptual framework defined in the

generic resource model from the same document [OMG01]. This allows the performance subprofile to take advantage of the mechanisms (e.g. modeling styles and stereotypes) that are provided for that framework.

Figure 2.4 The performance analysis domain model

The performance profile provides a notation for capturing performance requirements within the design context, for specifying execution parameters, which can be used by modeling tools to compute predictions, and for displaying performance results computed by modeling tools or found in testing [OMG01]. The OMG proposal for the performance profile was made public after the main work of this thesis was done. However, we have included in the thesis those elements from the profile that are necessary to annotate the activity diagrams with performance information.

2.4.2 Transformation of UML Models into Performance Models

UML is used for specification and design of object oriented systems. The basic feature of UML is to offer several diagrams for separating concerns of the different system views. This feature is helpful to derive a performance model that takes into account combined data from these diagrams. There are different approaches that can be taken in the transformation from UML to performance model. Some of these approaches will be presented as follows.

As a new research direction in the performance analysis area, in the past two years, Professor Dorina C. Petriu and her students have been deriving performance models from UML specifications by graph transformation. The approach is described in [Petriu98][Petriu99][Petriu01] etc. Firstly, the proposed methodology identifies frequently used architectural patterns [Petriu99]. The methodology developed a formal approach for generating LQN performance models from high-level software architectural patterns using Graph Grammars. This approach specifies the patterns using UML-Collaborations, which represents a society of classes, interfaces, and other elements together to provide some cooperative behaviors for a software system. Then, the structure of the performance models is automatically generated by applying graph transformations through the PROGRES tool to the high-level system architecture.

In [Amer01], it is proposed a method to determine the parameters of the performance model by graph transformation techniques, The first step is to automatically translate the UML sequence diagrams to activity diagrams, then use the activity diagrams to compute

the LQN parameters [Petriu01] [Amer01]. Graph transformation techniques were used since both the input and output models have well defined graphical representations.

There are several other approaches to generate performance models from software specifications. [Andolfi-00] uses software specifications specified by Message Sequence Charts to automatically build a QN model corresponding to the software architecture description. Another approach proposed in [Cortellessa+00] uses information from different UML diagrams, and produces a Software Execution Model then a System Model as in [Smith90]. The resulting model represents the whole software and hardware system. Due to space limitations, we will not describe other approaches here in detail.

2.5 Programmed Graph Rewriting System

A graph rewriting system is a tool that implements a graph grammar as a collection of rules to rewrite (i.e., transform) an input graph, including an algorithm for selecting the rules to be applied. The graph grammar is a specification of the family of graphs that can be constructed from a given graph instance. The algorithm guides the development effort. An example of such a system is the algorithmic attributed graph grammar system named PROgrammed Graph Rewriting Systems (PROGRES) that comprises a language and its corresponding toolset [Schuerr97].

The PROGRES tool has been used in the following areas [Schuerr97]:

- For specifying tools and data structures of an integrated software engineering environment that describes process modeling, version control, and configuration management tools.

- As the underlying principle of a fundamental new approach to diagram parsing.
- For defining the semantics of a visual database query language.

2.5.1 Components of PROGRES

The components of the PROGRES graph grammar are composed of typed nodes, labeled directed edges, a graph schema, the graph rewriting rules, and the algorithms that are used to select and apply graph rewrite rules [Schuerr90] [Schuerr94] [Schuerr97a] [Schuerr97b].

- **Node and Edge:** The primitive elements of a graph are the node and the edge. A node is considered to be a data structure that stores application information as attributes. A directed, labeled edge represents a binary relationship between the source and target nodes, with the edge label identifying the types of the relationship. Edges do not have attributes.
- **Graph Schema:** Application domain knowledge is captured as a specification that is also a graph, called a graph schema. The graph schema describes how a host graph can be constructed because it identifies what node-types and edge types can be connected, and the direction of the edge relationships between the nodes. It is very similar to a database schema defined in a binary entity relationship notation or object-oriented analysis class diagram.
- **Graph Rewriting Rules:** A host graph traversal and a complex graph update operation are understood as a graph rewrite rule. The two main graph rewriting mechanisms are the production and the transaction.

- The ***production*** is the most primitive graph rewrite operation. It modifies a fragment of the host graph, as illustrated in Figure 2.8.
- A ***transaction*** selects and applies one or more productions to rewrite a host graph. A transaction is atomic, either succeeding in entirety or leaving the host graph in its initial state. A transaction is made up of imperative control statements and declarative rules, to specify, select, and apply a set of graph rewrite rules.
- ***Algorithms:*** They are used to select and apply graph rewrite rules.

Figure 2.5 illustrates conceptually the application of the production rule from Figure 2.5(a) to the host graph in Figure 2.5(b). The production rule has a left-hand side (LHS) that shows a subgraph pattern to be replaced by its right-hand side (RHS). In this example, the node 1 from LHS will be kept in the RHS, whereas the node 2 will be deleted and replaced by two new nodes, 3 and 4. A subgraph that matches the left-hand side is found in the host graph (if more such subgraphs exist, one will be chosen in a nondeterministic fashion). The respective subgraph will be cut off from the host graph and replaced by the right hand side. Actually, the replacement procedure takes place as follows: the nodes to be kept are not eliminated together with their links with the rest of the graph. The nodes to be deleted are cut off, and their links are eliminated as well. The new nodes are added, linked by edges as shown in the right hand side.

a) A production rule with a left-hand side (LHS) and a right-hand side (RHS) subgraph pattern

b) Host graph before applying the production rule

c) Host graph after applying the production rule

Figure 2.5 Applying a Production Rule to a Host Graph

2.5.2 PROGRES Graph Schema

The graph schema is the basis of a PROGRES specification because it defines the node types that may be connected by each labeled edge, as well as node attributes [Schüerr97c]. The attributes of a node type or class are not shown in the graph schema figures for clarity.

- **Node type:** identifies an attribute's initialization value, as well as the manner in which derived attributes are (re)computed. A node type is a first order value, so it can be used as a parameter (e.g. in a function or a production) and it can be assigned as an attribute.
- **Node class:** is used for node type coercion. Node types with common properties have a common superclass. A node type can inherit the attributes of several node classes using multiple inheritance.
- **Edge:** A directed labeled edge represents a binary relationship between the source and target nodes, with the edge label identifying the type of the relationship. Edges do not have attributes.
- **Intrinsic relationships:** also called *edge types*, which are explicitly manipulated, and possess restrictions concerning the types of their source and targets.
- **Derived relationships:** which are defined by the means of a *path* or *restriction* expression. A *path* is represented by double arrows between two nodes, whereas a restriction is represented by double arrows pointing to one of the two nodes.
- **Intrinsic attributes:** which are defined for a particular set of node types and which are explicitly manipulated.
- **Derived attributes:** which are defined by means of directed equations and which may have different definitions for different node types.

The following Figure 2.6 represents the notation for defining the PROGRES graph schema [Schüerr97c]:

Figure 2.6 The Notation of PROGRES Graph Schema

- Rectangle boxes represent node classes, which are connected to their superclasses by means of dotted edges representing “is-a” relationships.
- Boxes with round corners represent node types which are connected to their uniquely defined classes by means of dashed edges representing “type is instance of class” relationships.
- Solid edges between node classes represent edge type definitions.
- Circles with a line segment attached to node classes represent attributes with their names above or below the connection line segment and their type definition nearby the circle.
- Circles with double line segment attached to node classes represent a derived attribute.

2.5.3 PROGRES Graph Rewriting Rules

In PROGRES, a graph rewrite rule is composed of transactions, tests, productions, restrictions, and path declarations [Schüerr97c]. These items are described below.

a) Test

Productions and tests are the parameterized, basic operations of the PROGRES language. A *test* is a parameterized graph query operation that is used to search for the existence (or nonexistence) of a subgraph pattern in a host graph. The host graph may contain additional edges not in the subgraph pattern. A test may have input and output parameters, such as attribute values, a node type, a node, or a set of nodes with common properties. Figure 2.7 shows a test.

Figure 2.7 An Example of a Test

b) Production

A *production* can be used like a test, but it has a second component for rewriting the graph. The left-hand side (LHS) of a production represents the pattern that is searched for and is described like a test. But, the right-hand side (RHS) of the production describes a replacement for the found subgraph pattern. A copy of the right-hand side graph is substituted for an occurrence of the left-hand side of the production. A production can alter the attributes of a node, add edges, delete edges, add nodes, delete nodes, and change the type of a node or edge. A single production can be used as a simple transaction [Schüerr97c].

A complex production or test can use restrictions or path expressions to define sets of nodes with specified attributes or complex relationships. A *restriction* is a negative (or positive) constraint on the attribute(s) or type of a single node. A *path expression* navigates from a given set of source nodes, along certain edges in a host graph, to return a set of target nodes. It can be viewed as either a derived binary relationship, or as a function that describes a graph traversal in terms of edges and other path expressions. A restriction or path expression succeeds if all positive node and edge patterns are found, while no match for the negative node pattern exists.

A simple example of a PROGRES production corresponding to the conceptual example from Figure 2.5 is given in Figure 2.8.

Figure 2.8 An Example of a Production

c) Transaction

A *transaction* is a graph transformation which either completes entirely (i.e. atomically) or aborts with no modifications. Figure 2.9 shows a transaction. A transaction has deterministic and non-deterministic control elements to govern the application of productions and tests.

```
Transaction AggragationAllSequenceNodes =  
    Loop  
        AggregatSequenceodes  
    End  
End:
```

Figure 2.9 An Example of a Transaction

d) Function

A function in PROGRES is like a procedure in other languages. It undertakes a piece of work which does not need a production. It can have parameters and a return value. Figure 2.10 shows a function.

```
function    max : (v1, v2 : integer) → integer =  
    [ v1 < v2 :: v2  
      | v1 ]  
end:
```

Figure 2.10 An example of a Function

c) Imported types and its functions

In PROGRES, a node attribute type is either a built-in type or an imported type. The built-in type can be boolean, integer and string; other attribute types can be imported from an external source library. Figure 2.11 shows an imported type and its functions.

```

from RealNumbers import

types
    Real;

functions
    R_IntToReal      : (integer) → Real,
    R_StringToValue  : (string) → Real,
    Real_ValueToString : (Real) → string,
    R_Zero           : (integer) → Real,
    PlusQuote        : (Real, Real) → Real,
    MinusQuote       : (Real, Real) → Real,
    MultiplyQuote     : (Real, Real) → Real,
    DividedQuote      : (Real, Real) → Real;

End;

```

Figure 2.11 An example of the Import Type and its Functions

Chapter 3 SPE Software Execution Model

This chapter presents in detail the Software Execution Model built in the first step of SPE and its analysis, as proposed in [Smith90]. The purpose is to present clearly the starting point and the problem solved in this thesis. The software execution model is presented together with a case study to show how the model is used. Since this model was proposed more than ten years ago, it was not designed for object-oriented software. In this thesis, we propose to use UML activity diagrams to represent SPE software execution models for both object-oriented and non object-oriented systems.

3.1 Software Execution Model Notation

The original notation for the Software Execution Model proposed by Smith in [Smith90] is based on elementary graph theory [Berge58], and is named an “execution graph”. An execution graph will be created by the modeler to show the execution structure for a workload scenario. In the execution graph, a node represents the execution of a certain action performed by a component of the system. A component is a collection of program statements, procedures, and abstract machine calls that perform a logical function in the software design. Control paths connecting the components are represented by arcs. The traversal of an arc represents a switch to the next execution step in the execution structure. It may also imply a context switch, such as a procedure call or an operating system call, if the execution steps are not contained within the same program or procedure.

In order to represent an execution model, the graph notation is summarized in Table 3.1 [Smith90].

Name	Symbol	Represents
Basic node		Functional component at current level of refinement
Expanded node		The function has been refined; details are in a subgraph.
Repetition node		Subsequent nodes repeat n times; the last node in the loop has an arc to this node.
Case node		Identifies nodes that are conditionally executed and their execution probabilities.
State identification node		Shows lock-free, fork-join, send-receive, and acquire-release processing events.
Split node		Shows the initiation of concurrent processes that need not join.
Arc		Flow passes from the origin to the destination
Call-return arc		Flow passes to the destination and returns to the origin node upon completion.
Driver arc		Same as above, except that control returns to the driver. X.
Dummy arc		No processing time is associated with the arc; they may be bi-directional.

Table 3.1 Execution Graph Notation Summary

3.1.1 A Case Study

In this section a case study originally described in [Smith90] is presented. The case study examines the design of an interactive system to support an engineering Computer-Aided Design (CAD) activity. Some computer graphics are used to construct a model of a proposed aerospace vehicle composed of nodes and beams, to store the model in a database, and to interactively assess the design's correctness, feasibility, and preferability. A scenario named DRAWMOD is chosen as an example. This example describes about a request from a user at a graphics terminal to draw an aircraft model previously stored in the database. It is used to view the model data for correctness and completeness.

Since SPE concerns quantitative analysis, realistic assumptions are made about the size of the problem and the performance requirements [Smith90]. The aircraft model consists of 2050 beams, and 1500 nodes. An interactive transaction is used to support the analysis of aircraft designs. A large amount of data needs to be retrieved and a large graphical display is required. The performance requirement for the DRAWMOD scenario is 10 seconds execution time.

According to [Smith90], the execution of the DRAWMOD scenario can be described as follows. When the request is received from the terminal, the DRAWMOD program is initiated. The beam definition and node location tables are OPENed. The database FIND locates all beam rows for the specified model id. The database DBSORT sorts the qualifying records into ascending sequence on the beam number.

For each beam the following operations are performed:

- **The database function RETRIEVE obtains the beam data.**
- **The two node numbers in the beam data are extracted.**
- **The database function FIND locates the node rows where the model_id number matches the user's input and the node numbers match those from the beam row.**
- **For each node, a database RETRIEVE obtains the node coordinates.**
- **The x, y, and z coordinates of each node are extracted and used to DRAW the graphic output.**
- **Upon completion, the beam definition and node location tables are CLOSED.**

Figure 3.1 The Execution Graph Model of the DRAWMOD Scenario (from [Smith90])

Figure 3.2 The Execution Graph Models of the EDMS. DBC processing is in (a), the Find routine is in (b), the Retrieve routine is in (c), DB_MP Initiation is in (d), and DB_MP Termination is in (e) (From [Smith 90])

The case study execution graph is described for the specified workload scenario in Figure 3.1. The DRAWMOD driver is the first node. It calls EDMS routines to retrieve the data for drawing the requested model. DRAWMOD first calls DBOPEN to open the database tables. (DBOPEN's processing details are not part of the case study, so it is a basic node.) Next, DRAWMOD calls FIND to locate all the beams for the specified model. (FIND and RETRIEVE are expanded nodes because their processing details are in the EDMS subgraphs.) Next DBSORT puts the beams into ascending sequence. Then the steps in the outer repetition loop execute once per beam, or approximately 2050 times.

Figure 3.2 shows the EDMS execution graphs. The top-level graph in (a) shows the general database control (DBC) processing. "DB_MP initiation" receives the request and routes it to the database control. The DBC driver calls "parse command," then calls "check access rights," and finally calls the appropriate routine to process the request. The case node shows the request processing currently defined. The Find and Retrieve routines are represented by expanded nodes; their subgraphs are in 3.3(b) and (c), respectively. The execution probabilities are data-dependent variables; values will be assigned to the variables in the graph analysis step. Upon completion, DBC transfers control to "DB_MP termination".

In Figure 3.2, the FIND and Retrieve subgraphs are described in (b) and (c). The case node for the "buffer checker" is in (c). When the desired data is not found in a buffer, the "Read (data)" component executes. "Read (data)" also has a data-dependent execution probability.

Parts (d) and (e) of Figure 3.2 show that “DB_MP initiation” obtains requests for EDMS functions, allocates memory for the process, RECEIVES, and interprets the requests. In this scenario, the request is for DBC processing. When the EDMS function completes, “DB_MP termination” formats the reply, SENDs it to DRAWMOD via SYS_EXEC, and frees memory.

Figure 3.3 shows the communication execution graphs; it also shows how the previous graphs relate to one another. When DRAWMOD requests an EDMS function, the process depicted in 3.3(a) executes. When DRAWMOD is coded, it will contain instructions that format the request, SEND the request to SYS_EXEC, and RECEIVE the reply when it is available. Figure 3.3 (b) shows the SYS_EXEC loop. When activated, it checks to see if requests are pending. When none are pending, it deactivates (sleeps) for a predetermined period of time. As long as the requests are pending, it processes them until finishing. Figure 3.3 (c) shows that Switchboard (Receive) gets the EDMS message, packages and prepares it for routing (in this case to DB_MP), updates the task schedule to activate DB_MP, and Switchboard (Send) forwards the message to DB_MP. Similarly, SYS_EXEC is invoked when the EDMS processing completes. SYS_EXEC receives the completion message, determines the destination (DRAWMOD), packages it, adjusts the schedule (to reactivate DRAWMOD), and Switchboard (Send) forwards the message.

Figure 3.3 The Execution Graph Models of the SYS_EXEC Communication Processing. The EDMS call processing is in (a), the SYS_EXEC loop is in (b), and the SYS_EXEC processing is expanded in (c).

3.2 The Best Time Analysis for the Execution Graph

After the execution graph is built, the modeler will associate to each node in the graph quantitative resource demands, such as CPU execution time, number of visits to different I/O devices, etc. Eventually, the resource demands can be converted into a time value. Smith proposed in [Smith90] a set of execution graph reduction rules that can be used to aggregate the times required for different nodes into a single time per graph. This represents the response time for the corresponding scenario, assuming that there is no contention in the system (named also “best-case” execution time).

Algorithm for the Reduction of Execution Graphs

Figures 3.4 to 3.7 represent the reduction rules for the execution graph introduced in [Smith90] for different graph structures: sequence, repetition, conditional, and parallel execution.

Figure 3.4 Sequential Path Reduction Rule

Figure 3.5 Repetition Path Reduction Rule

Figure 3.6 Condition Path Reduction Rule

Figure 3.7 Fork and Join Path Reduction Rule

The reduction algorithm examines the graphs, identifies the basic structure, computes the “execution time” of the structure, and reduces the structure to a single “computed node”. By applying the algorithm, a complicated graph can be finally reduced to a single computed node with the “execution time” equivalent to the execution time of the whole graph before reduction. A general structure consists of sequences, loops, cases, and fork/join. The rule of computing an equivalent “execution time” of each component is as follows:

- For a sequence, as in Figure 3.4, the computed node “execution time” is the sum of the “times” of the nodes in sequence.
- For loop structures, as shown in Figure 3.5, the execution time is computed by multiplying the node “time” by the loop repetition factor.
- For a case node, as shown in Figure 3.6, the average execution time is used. The average time can be computed by multiplying each node’s “time” by its execution probability. The execution probability can be input by user, or based on a visit time for each branch and number of branches to calculate.
- For a fork and join path, as shown in Figure 3.7, the shortest path is used to compute its execution time. Note that the shortest path uses the longest of the concurrent paths.

To be more precise, let t_j be the “time” for node j . Iteratively apply the reduction rules to transform a graph into an equivalent with fewer nodes and arcs; continue until the graph is reduced to a single node. Then the average “time” for the original execution graph is the resulting time for the reduced graph. The basic graph reduction rules follow.

- **Sequential-Path Reduction Rule:**
$$t_1^{i+1} = \sum_{j=1}^n t_j^i$$

Begin with the iteration number, i , initially equal to 1. The sequential-path reduction rule states that the time for the computed node for the $(i+1)^{st}$ iteration is the sum of the times of the nodes in sequence in the i^{th} iteration (see Figure 3.4).

- **Repetition-Path Reduction Rule:**
$$t_1^{i+1} = n t_1^i$$

The repetition –path reduction rule multiplies the node time in the i^{th} iteration by the loop repetition factor (see Figure 3.5).

- **Conditional-Path Reduction Rule:** $t_1^{i+1} = t_0^i + \sum_{j=1}^n \rho_j t_j^i$

The conditional-path reduction rule multiplies each node time in the i^{th} iteration by its execution probability, and adds the sum to t_0 , the time for determining which condition holds (see Figure 3.6).

- **Fork and Join Path Reduction Rule:** $t_1^{i+1} = t_0^i + \max_{j=1..n} (t_j^i) + t_{n+1}^i$

The above formula used in [Smith90] for shortest-path analysis assumes that the execution times of the parallel processes are deterministic, the parallel branches do not interfere with one another, and they join when the last one completes (that is, there is no additional synchronization delay). The time interval between the fork and join is thus the time of the longest of the concurrent paths in Figure 3.7. Note that this formula, which is applied only in the first SPE step, does not take into account execution time distributions other than deterministic, and ignores the effects of competition for resources between the parallel processes. The later aspect is taken into account in the second SPE step, when the software execution model is mapped onto a QN or LQN. In the second step, each parallel process will be mapped separately onto a performance model element, and the synchronization delay will be obtained by solving the performance model [Smith90].

3.3 Execution Time Calculation for the CAD Case Study

Applying the algorithm to the CAD case study presented in the previous section 3.1, we can calculate the execution time for the best-case scenario. Figure 3.8 shows the

resource requirements for each activity in the DRAWMOD scenario, as well as the total amount for the resource requirements. The reduction rules can be applied in two ways:

- a) Apply the reduction rules separately to the demands for different resources (e.g., separately for CPU time, File I/O, and Terminal I/O). As the reduction progresses, we obtain a vector of resource demands for the reduced nodes. This vector is converted eventually in time units.
- b) Convert all resource demands of a node in time units, then apply the reduction rule to the time of each node. For example, in order to convert the demand for Terminal I/O, in time units we multiply the number of operations with the average value of an operation.

In Figure 3.8, the reduction rules were applied separately as in case a).

Figure 3.8 The basic path reduction rules produce the total computer resource requirements for the DRAWMOD scenario

The basic path reduction rules produce the total computer resource requirements for the DRAWMOD scenario in Figure 3.8. The results of the data-dependent variable assignments are shown by their nodes. First the loop-reduction rule is applied to the innermost loop, getting the resource requirement vector (3810, 8.03, 0). In the vector, the first element is CPU usage, the second is File I/O usage, and the third is Terminal I/O usage. Next, the sequential-path reduction rule is applied to the beam loop body, getting the resource requirement vector (10315, 16.089, 1). The loop-reduction rule then gives (21145750, 32982, 2050). To get the total result of the computer usage for the scenario, we sum each activity with the sequential-path reduction rule. The total units of service for each computer resource i for the scenario obtained by applying the sequential-path reduction rule are as follows:

$$R_i = (21,843,733, 33,072, 2,052)$$

The average time to process a service request for computer resource i , is w_i . In Figure 3.8, $w_i = (0.0000006, 0.03, 0.01)\text{sec}$ (In [Smith90], each workstation has a CPU that processes 5 million instructions per second(MIPS), and up to 8 hard disks with an average service time of 30 milliseconds per I/O, so for CPU, it is 0.6×10^{-6} sec. File I/O is 0.03 sec. Terminal I/O is 0.01 sec.).

Based on the total execution time formula $T = \sum_{i=1}^k R_i w_i$, $i=1, \dots, k$, the total execution time for the DRAWMOD scenario is $T=1026$ seconds, or 17 minutes. It is clear that the 10 seconds performance goal will not be met.

This example shows that, even in the absence of contention, the best time analysis yields useful results for the analyst. More alternatives for this case study will be considered in chapter 6.

3.4 UML Solution for the Software Execution Model

From the case study, we know the execution graphs are used to model non object-oriented software, developed in traditional algorithmic paradigm. In this paradigm, the main software building block is the procedure or function. This view leads developers to focus on issues of control, and on the decomposition of larger algorithms into smaller ones. As the requirements change and the system grows, the systems built with an algorithmic focus are found to be very hard to maintain. When trying to apply SPE to object-oriented software, one realizes that the execution graphs are not very appropriate for directly representing the software execution model.

3.4.1 UML Sequence Diagram for the Software Execution Model

In a more recent work, Smith and Williams have extended the original SPE technique from [Smith 90] to be applied to object-oriented systems [Williams+98]. The authors have used a two-stage approach for building the software execution model:

- First, the scenarios were described by using Message Sequence Charts [ITUT-99], named also “extended sequence charts” (described in more detail in chapter 6)
- In the next stage, the sequence charts were translated by hand into execution graphs. The reason the execution graph notation is still used is because the reduction of these graphs, and their mapping to a QN model, were already implemented in a tool named SPE.ED™.

UML Sequence diagrams are good for representing scenarios because they focus on the time ordering of messages. Sequence diagrams emphasize the chronological course of

events, because the chronological course of communication between objects is indicated by the numbering of messages. However, they have some drawbacks: it is difficult to represent workflow because they lack a simple way to indicate iteration, conditional execution, and fork/join. For this reason, in [Williams+98] an extended sequence diagram notation is used.

3.4.2 UML Activity Diagrams for the Software Execution Model

In UML, an activity diagram is essentially a flowchart, showing the flow of control from an activity to an activity. It easily represents branching and parallel flows. In such a diagram, all or most of the states are activity states, and the majority of the transitions are triggered by the completion of activities. In this thesis we propose to use the activity diagram instead of the execution graph for scenario representation. Figure 3.9 shows an example of an Activity Diagram.

The next chapter will discuss in more detail the advantages and disadvantages of activity diagrams, and chapter 6 shows how we implemented the reduction of activity diagrams by graph transformations.

Figure 3.9 Activity Diagram Example

Chapter 4 Activity Diagram Metamodel

This chapter will focus on examining the activity diagram and its associated metamodel to explore the possibility of using the activity diagram for performance analysis purposes.

4.1 Describing Behaviour in UML

We considered the UML diagrams that model behavior with interaction diagram, statechart diagrams, and activity diagram.

- The *interaction diagrams* (Sequence and Collaboration) are excellent for representing inter-object behavior, but are less useful at showing what happens inside an object.
- The *statechart diagrams* show the internal object behavior, but are less effective in showing the inter-object collaboration.
- The *activity diagrams* are able to combine both inter and intra-object behavior, describing the system behavior at different levels of abstraction, from high-level view of scenarios to detailed views of an object operation.

We choose to use the following activity graph features from UML1.4 [UML 1.4]:

- separate swimlanes for each concurrent instance to show the actions performed by each instance ([UML 1.4], pg.3-166)
- object flow states to represent the content of messages ([UML 1.4], pg.3-168).

Figure 4.1 represents two alternatives for modeling the sending/receiving of an asynchronous message. In Figure 4.1.a, the ActionState that represents the sending of the message is followed by an explicit forking: one thread for the continuing execution of the sender, and the other thread for the message just sent. On the receiver's side, the message is accepted through a joining of the receiver's thread with the message thread. This model is very close to a Petri net model. The disadvantage of the approach shown in Figure 4.1.a is that the ObjectFlowState is not directly connected to the sending and receiving action states. Also, there are too many forks and joins in the model that may cloud the understanding of the diagram.

Figure 4.1.b is a variation on Figure 4.1.a: it uses the signal send/receipt symbols from UML1.4, p. 3-170. The advantage is that the sending and receiving actions stand out, making the diagram easier to read. However, the mapping of the "signal send" symbol given in UML1.4 p.3-172 should be changed to an ActionState instead of a SendAction on a transition. Also, this solution inherits the disadvantages from Fig.1a: the object flow is not directly connected with the action states that produce/take it as output/input, and there are too many fork/joins in the model.

Figure 4.1 Possible alternatives for modeling the sending/receiving of an asynchronous

In this thesis we use the alternative from Figure 4.1.a for representing the message flow between concurrent components.

A scenario can be represented in UML either by a sequence or by an activity diagram. A sequence diagram shows a series of messages exchanged by a selected set of objects in a temporally limited situation, with an emphasis on the chronological course of events. In the sequence diagram objects have vertical lifelines, and the chronological course of communication between objects is indicated by the message numbers. This highlights the chronological sequence of the messages. However, the sequence diagrams cannot clearly describe some cases. For example:

- When using the notation *[Condition] message()* for conditional execution, the extent of every conditional alternative is not always clear.
- When an asterisk (*) is placed in front of the message to indicate iterations, it is not clear what portion of the diagram has to be repeated.
- Parallel execution flows (fork/join) cannot be expressed easily in the UML sequence diagram.

It should be noted that these features are part of the Message Sequence Charts (MSC) which is an ITUT standard [ITUT-97]. In fact, William and Smith have used MSC in their work on SPE for object-oriented software [William +98] discussed in Chapter 3. However, we decided not to use MSC because we wanted to remain in the UML domain. Since the UML sequence diagrams have the disadvantage listed above, we decided to use activity diagrams for SPE software execution models.

The activity diagrams have some advantage over the sequence diagrams when modeling scenarios:

- Activity diagrams are more suitable for representing execution flows, because they express easily branching, fork/join, iteration, and nesting of activities.
- Activity diagrams support the description of parallel activities. For activity paths running in parallel, the order is irrelevant. They can run consecutively, simultaneously, or alternately.
- Activity diagrams are much better suited for expressing branching (or alternatives) than the sequence diagrams.

The activity diagram has a major disadvantage with respect to the sequence diagram: it does not show what objects are responsible for executing the actions. We can partially overcome this disadvantage by using swimlanes to express responsibilities [Fowler+99]. In this thesis we propose to use activity diagrams with swimlanes for representing the SPE software execution model. A reduction algorithm for activity diagrams is also introduced. However, we will discuss first the UML metamodel for activity diagrams.

4.2 Activity Diagram and Associated Metamodel Elements

The UML metamodel is the base of the UML semantic specification and is described with a subset of UML notations. The metamodel is described by using static diagrams with metaclasses that represent the UML modeling elements, such as classes, interfaces, states, actors, associations, etc. These metaclasses describe the properties and

associations of the modeling elements and their constraints. We need to understand the metamodel in order to understand what metaobjects are created for a given activity diagram, and what their relationships are. The reduction rules will manipulate these metaobjects.

The UML metamodel has many small static metaclass diagrams, where each diagram is used to represent a set of UML semantic specifications from a different view. We put all the metaclasses related to activity diagram into a single diagram (see Figure 4.2) to have all the elements on the same page. This figure includes elements from Action Graph, State Machine and Common Behavior from [UML 1.3].

Figure 4.2 UML metamodel for Activity Diagrams (it includes elements from Action Graph, State Machine and Common Behaviour diagrams)

4.2.1 Model Elements for Activity Diagrams

In the UML metamodel, the elements from Figure 4.2 are defined as follows according to [OMG-99].

- ***State Machine***

It specifies the behavior of various elements that are being modeled. For example, it is used to specify the behavior of individual entities or to define the interactions between entities. The State Machine formalism provides the semantic foundation for activity graphs. Activity graphs are simply a special form of state machines.

- * ***State***

A state is an abstract metaclass that models a situation during which some invariant condition holds. The invariant may represent a static situation such as an object waiting for some external event to occur. However, it can also model dynamic conditions such as the process of performing some activity.

State is a child of StateVertex.

- ** ***Composite State***

A composite state is a state that contains other state vertices (States, pseudostates, etc.). The association between the composite and the contained vertices is a composition association. So, a state vertex can, at most, be a part of one composite state.

A *substate* of a composite state is that any state is enclosed within a composite state. When a composite state is not contained by any other state, it is called a *direct substate*.

Composite State is a child of State.

***** Submachine State**

A submachine state is a syntactical convenience that facilitates reuse and modularity. The state machine that is inserted is called the *referenced* state machine, while the state machine that contains the submachine state is called the containing state machine. In effect, a submachine state represents a “call” to a state machine “subroutine” with one or more entry and exit points.

SubmachineState is a child of State.

****** Subactivity State**

A subactivity state represents the execution of a non-atomic sequence of steps that has some duration (i.e., internally it consists of a set of actions possibly waiting for events). That is, a subactivity state is a “hierarchical action” where an associated subactivity graph is executed. A subactivity state is a submachine state that executes a nested activity graph.

**** Simple State**

A SimpleState is a state that does not have substates. It is a child of State.

***** Action State**

An action state represents the execution of an atomic action, typically the invocation of an operation. An action state is a simple state with an entry

action whose only exit transition is triggered by the implicit event of completing the execution of the entry action.

***** *Object Flow State***

An object flow state defines an object flow between actions in an activity graph. An instance of a particular classifier, possible in a particular state, is available when an object flow state is occupied.

**** *Final State***

FinalState is a child of State. Final state is a special kind of state signifying that the enclosing composite state is completed. It cannot have any outgoing transitions.

*** *State Vertex***

A StateVertex is an abstraction of a node in a statechart graph. In general, it can be the source or destination of any number of transitions.

StateVertex is a child of ModelElement.

**** *PseudoState***

A pseudostate is an abstraction that encompasses different types of transient vertices in the state machine graph. They are typically used to connect multiple transitions into more complex state transitions paths.

PseudoState is a child of StateVertex.

□ *Initial*

An initial pseudostate represents a default vertex that is the source for a single transition to the default state of a composite state. There can be at most one initial vertex in a composite state.

□ ***join***

Join vertices serve to merge several transitions emanating from source vertices in different orthogonal regions. The transitions entering a join vertex cannot have guards.

□ ***fork***

Fork vertices serve to split an incoming transition into two or more transitions terminating on orthogonal target vertices. The segments outgoing from a fork vertex must not have guards.

□ ***junction***

Junction vertices are semantic-free vertices that are used to chain together multiple transitions. They are used to construct compound transition paths between states. For example, a junction can be used to converge multiple incoming transitions into a single outgoing transition representing a shared transition path (that is known as a merge). Conversely, they can be used to split an incoming transition into multiple outgoing transition segments with different guard conditions. This realizes a static conditional branch. Static conditional branches are distinct from dynamic conditional branches that are realized by choice vertices.

□ ***choice***

Choice vertex is a dynamic conditional branch, in which a choice is made according to the dynamic evaluation result of the guards of its outgoing transitions. It allows splitting of transitions into multiple outgoing paths, so that the decision about which path to take may be a function of the results of prior actions performed in the same run-to-completion step. If more than one of the guards evaluates to true, an arbitrary one is selected. (If none of the guards evaluates to true, then the model is considered ill-formed. To avoid this it is recommended to define one outgoing transition with the predefined “else” guard for every choice vertex)

- ***Guard***

A guard is a boolean expression that is attached to a transition as a fine-grained control over its firing. If the guard is true at that time, the transition is enabled; otherwise, it is disabled. Guard is a child of ModelElement.

- ***Transition***

A transition is a directed relationship between a source state vertex and a target state vertex. Transition is a child of ModelElement.

- ***Partition***

A partition is a mechanism for dividing the states of an activity graph into groups. Partitions often correspond to organizational units in a business model, and help

allocate properties and actions for various purposes. In our work, we will use a partition for the activities performed by each concurrent component.

- ***Activity Graph***

An activity graph is a special case of a state machine that defines a computational process in terms of the control-flow and object-flow among its constituent actions. It does not extend the semantics of state machines in a major way, but it does define shorthand forms that are convenient for modeling control-flow and object-flow in computational and organizational processes.

The primary purpose of activity graphs is to describe the states of an activity or process involving one or more classifiers. Activity graphs can be attached to packages and classifiers.

- ***Swimlanes***

A swimlane maps into a Partition of the States in the Activity Graph. A state symbol in a swimlane causes the corresponding State to belong to the corresponding Partition.

4.2.2 Execution Time Specification

The standard UML models does not contain all information required for performance analysis. However, the OMG performance profile [OMG01], which extends the UML metamodel with standard extension mechanisms (i.e. tagged values and stereotypes), introduces extensions necessary for performance analysis. In particular, we are interested

here in the stereotype PStep from Figure 2.4, that introduces a number of tagged values useful for performance analysis, such as *probability*, *repetition*, *executionTime*. The stereotype Pstep can be applied to Action or ActionState, which allows us to augment the activity diagrams with the desired performance information.

4.3 Metaobjects Diagrams

This section describes the metaobject diagrams created by a UML tool for different activity diagrams. We need to understand what metaobjects are created for various structures (such as sequence, conditional execution, loop, fork/join) in order to define the graph reduction rules in the following Chapter.

Figures 4.3 to 4.6 show different activity diagram structures and their corresponding metaobject diagrams.

Activity Diagram Notation

Corresponding UML Metamodel Objects

Figure 4.3 Sequence Structure

Activity Diagram Notation

Corresponding UML Metamodel Objects

Figure 4.4 Branch-Merge Structure

Figure 4.5 Fork-Join Structure

Activity Diagram Notation

Corresponding UML Metamodel Objects

We can represent loops in activity diagrams as follows

Figure 4.6 Loop Structure

The activity diagram notation provides for the use of ObjectFlowStates that models the data flow between activities. An ObjectFlowState is drawn as a rectangle connected by dotted lines with the activity that produces the object as output and with the activity that takes the object as input. In our reduction algorithm, we will treat the ObjectFlowState like any other State. We may even associate a time with an ObjectFlowState that represents the delay of the respective object on the communication link from source to destination.

In this chapter, we discussed the UML metamodel for activity diagrams. In the next chapter, we will use the PROGRES language to implement the proposed approach for reducing activity diagrams by graph transformations.

Chapter 5 Activity Diagram Reduction by Graph Transformations

As described before, we use PROGRES [Schüerr94] [Schüerr97a] [Schüerr97b] [Schüerr97c], a graph transformation tool for the implementation of the UML activity diagram reduction. A PROGRES program contains a set of rules that transforms one instance of a given class of graphs into another instance of the same class of graphs, without distinguishing terminal and nonterminal nodes [Schuerr97]. In this chapter we will focus on the PROGRES features needed for the activity diagram reduction. A more general description about the PROGRES is given in Chapter 2.

5.1 Graph Transformations

A graph grammar is used for transforming a model (described as a graph) from its initial domain into the target domain; the transformation result is an output graph model [Schüerr97c]. The approach is to transform a host graph model, in small steps, until the transformation graph complies with the target domain's graph grammar. This translation is a rule-based graph transformation with a set of graph transformation rules and a control algorithm for selecting the rule(s) to be applied. To accomplish this, a synthesized graph syntax is developed to combine the input domain's graph syntax and the target domain's graph syntax. A set of graph transformation rules, using the synthesized graph syntax, is specified to convert the terminal symbols (i.e. sub-graphs of node types) from the initial domain into terminal symbols of the target domain (i.e., a replacement sub-graph). A

control algorithm governs the application of these transformation rules to prevent or manage conflicts for the rule selection. The principle of graph transformation is shown in Figure 5.1.

Figure 5.1 Model Transformation by using Graph Grammars

In our case, the Input domain and Output domain are identical, both representing activity diagrams.

Three items are used to exploit a PROGRES implementation: the graph grammar itself, a graph transformation system which implements the graph grammar, and methodology for the use of both. The *graph grammar* is a specification of the family of graphs that can be constructed from a given graph instance (called the host graph). A *graph transformation system* is a tool that implements a graph grammar as a collection of rules to transform an input graph, including an algorithm for selecting the rules to be applied. The *methodology* guides the development effort. These three components are available for the algorithmic attributed graph grammar approach, which is implemented by the PROGRES language and its corresponding toolset [Schuerr90].

5.2 PROGRES Graph Schema

PROGRES provides a language feature to define the static properties of graphs that take in the form of a graph schema which is conceptually similar to a database schema. The graph schema definition part of a PROGRES specification enables us to specify static properties of any class of directed, attributed, node and edge labeled graphs. The PROGRES Schema, which is used in UML activity diagram reduction, is obtained from the UML metamodel. The mapping between the UML notation and PROGRES schema notation is shown in Table 5.1.

5.2.1 Mapping the UML Notation to PROGRES Schema Notation

The inheritance relationship from the UML metamodel can be represented by dashed edges or dotted edges in the PROGRES Schema. The class in the UML metamodel can be represented by a node class or by a node type in PROGRES Schema. A PROGRES node class plays the same role as an UML abstract class from which no objects (nodes) can be instantiated. A PROGRES node type is like a UML concrete class from which objects (nodes) can be instantiated. The associations, aggregations, and compositions in the UML metamodel can be represented by edges in PROGRES. All of the extended representations are listed as follows.

In the UML metamodel, the inheritance relationship between a subclass and its superclass is represented by a line with a triangle arrow as Table 5.1. However, in the

PROGRES, the inheritance relationship is represented with both dotted edges and dashed edges as shown in Table 5.1.

UML Metamodel	PROGRES Schema
Abstract class	Node Class
Concrete class	Node Type
Inheritance Relationship
Association
Aggregation
Composition
Two Role names on association <div> <div>+ source</div> <div> <div>1</div> <div>•</div> </div> <div>+outgoing</div> </div>	One edge name <div> <div>←</div> <div>Source (or outgoing)</div> </div>
One Role name <div> <div>+trigger</div> <div> <div>0..1</div> <div>•</div> </div> </div>	One Role name <div> <div>+trigger</div> <div>_____</div> </div>

Table 5.1 UML Metamodel vs. PROGRES Schema Notation

In PROGRES, a dotted edge with an arrow is used to connect a node class to its superclass node to represent “is-a” relationship. A superclass could have one or many subclass(es).

In PROGRES, a dashed edge with an arrow is used to connect a set of node types to their uniquely defined classes to represent “type is instance of class” relationships. A node class could have one or many node type(s). For example, in Figure 5.2(b), the PseudoState has nine node types.

In UML metamodel, the class is represented by a rectangle box as shown in Table 5.1. However, in the PROGRES Schema, the class is used instead of a node class that has a non-leaf, or a node type that has leaf as in Table 5.1.

In PROGRES, a node class is represented by a rectangular box which is connected to its superclass. A node class is not a leaf in the PROGRES Schema.

In PROGRES, a node type is represented by a box with round corners which is connected to its uniquely defined class. A node type is a leaf in the PROGRES schema, or a node class could have one or many node type(s).

The associations, aggregations, and compositions of the UML metamodel are represented by an edge or edges in PROGRES Schema as in Table 5.1.

If the association, aggregation, and composition are navigable in the UML metamodel, we keep the same direction of the arc in the PROGRES Schema as in Table 5.1.

In Table 5.1, the name of the PROGRES edge types is related with the role name of UML association ends. If an association arc in UML has two role names with one at each end, we will choose one of them to represent the edge type in the PROGRES schema. If only one association end is named in the UML metamodel, the same name is used for the edge type in the PROGRES schema. For example, in Table 5.1, there is a type whose name is trigger in the UML metamodel. It is represented as an edge type with the same name in PROGRES.

5.2.2 PROGRES Schema Transformation from UML Metamodel

As mentioned above, a graph schema is the basis of a PROGRES specification, and it defines the node types that may be connected by each labeled edge, as well as node attributes. The graph schema, shown in Figure 5.2(b), for the transformation from the UML activity diagram defines the types of nodes and edges allowed in an input graph (an activity diagram), an output graph (reduction activity diagram) and an intermediary graph (a combination of both input and output graphs). The input schema does not capture all the richness of UML, but only those elements that are necessary for reducing an activity diagram.

In UML, each diagram can be described in terms of their constituting elements and in terms of the relationship existing between these elements. Figure 5.2 (a) shows a UML metamodel description focused on activity diagrams. To reduce activity diagrams, the majority of elements in Figure 5.2(a) are mapped to elements of PROGRES schema in Figure 5.2(b), which represents the PROGRES notation for the UML metamodel for

activity diagrams. We note that both Figure 5.2(a) and Figure 5.2(b) contain some nodes in gray; these nodes do not have a corresponding counterpart in the other figure. For example, *SynchState* node from Figure 5.2 (a) is not mapped in Figure 5.2(b). In the same way, *LoopBegin* node from Figure 5.2 (b) does not have a corresponding node in Figure 5.2 (a). In our examples, the PROGRES input graph for a given system is directly generated by a PROGRES program according to the schema from Figure 5.2(b).

The advantage of basing the transformation on UML activity diagram is that the higher-level of abstraction artifacts greatly simplify the graph schema and the transformation process. The disadvantage is that these artifacts have to be pre-identified and represented in the schema and transformation rules, which limits the extensibility of the transformation process.

Figure 5.2 (a) UML metamodel for Activity Diagrams (it includes elements from Action Graph, State Machine and Common Behaviour diagrams)

Figure 5.2(b) PROGRES Schema for Activity Diagrams

As we know, elements in activity diagrams are represented in the form of *nodes* and *edges* in PROGRES. There are two kinds of nodes: *node types* and *node classes*. The node types are those which determine the static properties of their node instances, while the node classes are definitions of common node type properties. The relationship between node types and node classes is similar to the relationship between derived classes and their abstract base classes.

Figure 5.2(b) shows the PROGRES Schema for an activity diagram, and illustrates the inheritance relationship between different node classes and states. As we can see from Figure 5.2(b), *MODEL_ELEMENT* is the root node class. *STATE_VERTEX*, *TRANSITION*, *ACTION*, *STATE_MACHINE*, *PARTITION*, and *GUARD* are node classes that inherit from the root node class. *PSEUDO_STATE*, another node class, inherits from *STATE_VERTEX*. The inheritance relationships also exist between *STATE* and *COMPOSITE_STATE*, *SIMPLE_STATE*, and *FINAL_STATE*. The node class *SUBMACHINE_STATE* inherits from *COMPOSITE_STATE*, and *SUBACTIVITY_STATE* inherits from *SUBMACHINE_STATE*.

Each node class may have different node types (As mentioned before only the node types can be instantiated in the PROGRES):

- *PSEUDO_STATE* node class has nine different node types as follows:
Initial, Begin, End, fork, join, branch, merge, loopBegin, and loopEnd.
- *TRANSITION* node class has *Transition* node type.
- *ACTION*, *GUARD* and *STATE* node class has *Action, Guard, and State* node type, respectively.

- *SIMPLE_STATE* node class has *ActionState* and *ObjectFlowState* node types
- *SUBMACHINE_STATE* has its node type, *SubactivityState*.
- *ActionState* and *ObjectFlowState* node types belong to *SIMPLE_STATE* node class.

There are two types of edges which define intrinsic relationships and derived relationships in PROGRES. The intrinsic relationships are explicitly defined between sources and targets that are directly connected, but the derived relationships define a path expression between a source and a target that are not directly connected.

For example, in Figure 5.2(b), we can see edge types named *target* and *source* between *STATE_VERTEX* and *TRANSITION* node class. Other edge types are: *entry*, *guard*, *top*, and *iteration*.

A node's attributes serve as local variables whose values are stored intrinsically or computed at run-time. For example, *executionTime* is a node's attribute for *Action* node type. The built-in PROGRES attribute types are *boolean*, *integer*, and *string*; other attribute types can be imported from an external source library. For example, the *Guard* node type in Figure 5.2(b) has an attribute whose name is *probability* of real type, which needs to be imported by user. An intrinsic attribute, such as *executionTime* in *Action* node type, can be read and set during a graph modification.

A derived attribute's value is specified as a function that is calculated at run-time, whose parameters are the structure of the graph and the attribute value(s) of any nodes(s) in the graph. Derived attribute evaluation follows a lazy strategy, where read access of a

derived attribute forces all attributes they depend on to be evaluated [Kiese+]. Once a derived attribute is evaluated its value is stored. As long as the attribute is neither directly nor indirectly affected by modifications of the graph, the value remains valid. Note that in the proposed Progres schema, the *Guard* node type not only has a *probability* attribute for the branch structure, but also a *number of iterations* (loop repetition factor) for the loop structure.

5.3 PROGRES Graph Transformation Rules

In this section, we describe the graph transformation rules for reducing activity diagrams.

5.3.1 General Transformation Principles for the Reduction of Activity Diagrams

When a graph transformation rule is applied in our implementation for an activity diagram reduction, the graph transformation rule should obey following conventions.

- A common input parameter to all graph transformation rules is the node in the host graph that is potentially the beginning of a graph transformation rule match (called a *trigger node*).
- All graph transformation rules return the new trigger node to be considered next, as well as any service periods to be processed at a later time.

- When a graph traversal is needed, the graph transformation rules will seek the shortest causal path using a depth-first search on the task event graph, crossing over to different tasks only when necessary.

In the activity diagram reduction, the general transformation principles are as follows:

- The reduction of each basic execution structure has its own reduction rule. The basic structures are sequences, loops, branches-merge, and fork-join. The host graph is examined, a basic structure is identified automatically by PROGRES, its execution time is computed, and the structure is reduced to a node. The advantage of using a graph-rewriting tool such as PROGRES is the fact that the programmer does not have to worry about identifying a subgraph in the host graph that matches the left-hand-side of a production rule, because this task is performed automatically by the tool.
- Continue to reduce the structures until the graph is reduced to a single node. The execution time for the single node is the final result, which represents the best-case response time for the scenario.
- If there is a nested execution structure inside another structure, reduce the nested execution structure first.
- When there is no basic execution structure left in a given activity diagram, the reduction work is done.

5.3.2 Productions Rules

A *production* has a left-hand side sub-graph (LHS) and right-hand side sub-graph (RHS). The LHS is used to match a portion of the graph using the graph schema. A LHS match is attempted relative to the trigger node supplied by the control algorithm for a graph transformation. For this reason, each production has a *glue node* that is bound to the *trigger node* when a match is attempted. Once the glue node is bound to the trigger node, the LHS matching process continues by attempting to bind the LHS edges and remaining nodes to other nodes in the graph, since all nodes are structurally related to the glue node. Once the LHS nodes and edges are bound in the graph, any supplied operational constraints are tested for validity. A LHS match succeeds if all of the nodes and edges occur in the graph and all of the operation constraints are satisfied. If the LHS is matched, then the graph is rewritten, according to the RHS sub-graph. The LHS of a production includes all of the nodes and edges that are involved in an interaction and are modified by the RHS. It can also include nodes which are not modified but are used to anchor the replacement sub-graph of the RHS [Schüerr97c].

In a production, the RHS removes or replaces the graph node types and edges identified by the LHS. Only the sub-model node types, edge types, and node classes are allowed in the RHS.

In the reduction of activity diagrams, the productions are defined for each basic execution structure. There are four kinds of basic execution structures in activity diagrams, which are sequence, branch-merge, loop, and fork-join structures. The productions are examined in detail, shown through the sections 5.3.2.1 to 5.3.2.4 below.

5.3.2.1 Sequence Structure Production Rule

Figure 5.3 shows the production rule for reducing a sequence structure. The structure has more than two nodes which are connected in sequence. The following is the steps for the transformation of the sequence structure from an input to an output graph. The transaction to reduce any number of sequential nodes consists of the following basic operations:

1. Finding two State nodes that are connected in sequence
2. Replace the two nodes with a reduction node.
3. Repeating steps 1 and 2 until no sequence structure is left.
4. After the reduction, the execution time of the single node is the sum of the reduced nodes.

production ReductionSequencePathNodes =

::=

```
transfer      10'.name := ('3.name) & ", " & ('6.name);  
                11'.name := "NewAction(MixAction4and7)";  
                12'.name := "NewTransition";  
                11'.executionTime := ('4.executionTime) +  
                                     ('7.executionTime);  
  
end;
```

Figure 5.3 Production Rule for the Reduction of a Sequence Structure

We can see in Figure 5.3 that two sequential states (nodes '3 and '6 in the LHS) connected by a transition (node '5) are all replaced by a single state (node 10' in the RHS).

5.3.2.2 Branch and Merge Structure Production Rule

Figure 5.4 and Figure 5.5 show an activity diagram with branch(es) and merge(s) structure. Based on the basic reduction rules, the structure transformation from an input to an output graph has following basic operations:

1. Finding two nodes that are connected by branch structure
2. Replace the nodes with a branch reduction node.
3. Repeat steps 1 and 2 until only one branch is left
4. Replace the one branch structure nodes with a final reduction node.
5. Execution time of the reduced structure is calculated as an average execution time by a formula shown in Figure 3.6.

The probability can be input by the user, or calculated based on the number of visits for each branch and number of branches.

The reduction of a structure with multiple branches requires two production rules, The first, shown in Figure 5.4 reduces two branches into one. The second production shown in Figure 5.5 reduces the last branch by eliminating the *branch* and *merge* nodes. The result is a single state node for the whole structure.

Step One: ReductionMoreThanTwo BranchesMergesPathNodes

production ReductionMoreThanTwoBranchesMergesPathNodes =

::=

```

Transfer      19'.name := "NewBranchNodes_A";
                21'.name := "AggCasePathNodes";
                23'.name := "NewBranchNodes_B";
20'.Probability:=('7.Probability)+('10.Probability);
22'.executionTime:=(((('7.Probability)*('12.executionTime))+
(('10.Probability)*('14.executionTime)))/
(('7.Probability)+('10.Probability));

end;

```

Figures 5.4 Production Rule for Reduction of More Than Two Branches

Step Two: ReductionOneBranchMerge PathNode

production ReductionOneBranchMergePathNodes =

::=

```

transfer      14'.name := "AggCasePathNodes";
                15'.executionTime := ('10.executionTime);

end;

```

Figure 5.5 Production Rule for the Reduction of the last branch of a branch/merge structure

5.3.2.3 LoopBegin and LoopEnd Structure Production Rule

Figure 5.6 shows an activity diagram with a loop structure, i.e. LoopBegin and LoopEnd Structure. Based on basic reduction rules, the structure transformation from an input to an output graph has the following basic operations:

1. Finding a loop structure
2. Replace the loop with a branch reduction node

3. Repeat steps 1 and 2 until a single branch structure is left
4. Execution time of the reduced structure is the average execution time of each node's execution time multiplied by the loop repetition factor.

production ReductionLoopPathNodes =

::=

```

transfer    15'.name := "AggLoopPathNodes";
              16'.executionTime :=
                (('13.numberOfIteration) *
                ('9.executionTime));

```

end;

Figure 5.6 Production Rule for the Reduction of a loop structure

5.3.2.4 Fork and Join Structure Production Rule

Figure 5.7 shows an activity diagram with a Fork and Join structure in which at least two nodes are connected like a branch structure. Based on the basic reduction rules, the structure transformation from an input to an output graph has the following basic operations:

1. Finding two nodes that are connected by fork and join structure

2. Replace the two nodes with a reduction node
3. Repeat steps 1 and 2 until no more fork and join structures are left

In the structure, we consider the shortest path for calculating the execution time of a scenario. The shortest path analysis assumes that the concurrent processes do not interfere with one another and that the processes join when the last one completes, that is there is no additional synchronization delay. The execution time interval between the fork and join is thus the time of the longest of the concurrent paths [Smith90].

production ReductionParallelPathNodes =

```

transfer      17'.name := "AggForkNode";
                 18'.executionTime :=
                  max ( ('9.executionTime),
                        ('12.executionTime) );

end;

```

Figure 5.7 Production Rule for the Reduction of a Fork/Join Structure

5.3.3 Transaction for Activity Diagram Reduction

A *transaction* is a graph transformation algorithm that either completely succeeds or fails. A transaction is programmed like a procedure except that it can include productions or other transactions. In the activity diagram reduction, a transaction is executed for every execution structure found in the input activity diagram. The transformation process ends when no basic structure is left in the input graph.

As shown before, there are four basic execution structures (sequence, branches-merge, loop, and fork-join). In the sequence structure, the transaction is the following. The Loop statement exits automatically when no subgraph matching the LHS of the production rule contained in the loop is found in the host graph.

```
Transaction  Reduction All Sequence Path Nodes =  
    Loop  
        Reduction Sequence Path Nodes;  
    End  
End;
```

Figure 5.8 Transaction for the Reduction of All Sequence Path Nodes

Figure 5.9 illustrates a host graph with five sequence structures. The transaction from Figure 5.8 will reduce all of them, one by one. Since PROGRES chooses nondeterministically a subgraph matching the LHS of the production, the order in which these structures will be reduced may vary from execution to execution. However, no matter in which order these sequence structures are reduced, we will obtain the same result, which is the last graph in Figure 5.9. This occurs because the sequence structure

reduction adds the service times. If several sequence structures follow each other, the reduction will eventually generate a final node whose time is the sum of the times of the reduced nodes.

Figure 5.9 The reduction of a graph with several sequence structures

The transaction from Figure 5.9 will stop where there is no sequence structure left. During the execution of this transaction, no other type of production rule will be applied.

Figure 5.10 gives a different reduction order for the sequence structures nested on alternative branches shown in the last steps from Figure 5.9. The conclusion is that the resulting graph is the same. The order of reducing the sequence structures does not affect the correctness of the result as long as only sequence structures are reduced until none is left. The reduction of a branch structure is possible only when all its branches are reduced to a single node. The same is true for loops and fork/join structures.

Figure 5.10 A different reduction order for the last steps of **Figure 5.9**

The transactions for the other three execution structures have a similar transformation algorithm. It can be shown that for each algorithm the final result does not depend on the order of reduction, as long as we do not mix the reduction of different kinds of structures.

5.4 Control Algorithm

The algorithm is a controller, which manages the order in which the transactions are applied. In the algorithm, the graph transformation rules are selected and applied to a task and operation event graph. There are two components in the algorithm. One is an initialization component, another an analysis component. The general operation of the algorithm is to consider each task's service period in entirety before analyzing another task's service period. This is done to ensure that events are processed in a manner consistent with operation causality, so that events are not considered out of order which could cause interpretation difficulties [Schüerr97c].

The algorithm for the reduction of activity diagrams is shown in the following pseudo code:

```
Transaction Reduction Activity diagram
Create an input graph for an Activity diagram;
Begin
    While (still have two or more nodes left)
        Loop
            Loop
                Reduction Sequence Path Nodes;
            End
        End
    End
```



```

    Loop
        Reduction More than two Branches Path Nodes;
    End
    Loop
        Reduction One Branch Path Nodes;
    End
    Loop
        Reduction Loop Path Nodes;
    End
    Loop
        Reduction Parallel Path Nodes;
    End
End
End

```

Figure 5.11 The Algorithm for the Reduction of an Activity Diagram

In the algorithm, the initialization component begins by taking all of the external nodes and inserting them into the trigger node list. The head of the trigger node list is made the trigger node, and the analysis process begins [Schüerr97c].

The analysis component of the algorithm consists of the outermost loop and five inner loops inside the loop. The outermost loop ensures that all of the nodes in the trigger node list are processed. These inner loops process all the nodes in a service period of a task, sequentially making each node a trigger node, and skipping over those nodes that were part of a previous match. Once a service period has been fully analyzed, the head node on the trigger node list is made the new trigger node and the analysis continues.

In summary, the algorithm in Figure 5.9 does the following:

- The PROGRES program creates a input graph from transaction CreateMixPath
- Then it performs transformation of each execution structure in a loop. In Figure 5.9, each pattern, corresponding to a execution structure, is executed in the following order:
 - all the sequence structures (first inner loop)
 - all the branch structures (second inner loop)
 - all the loop structures (third inner loop)
 - all the fork/join structures (forth inner loop)
- After completing these steps, the outer loop is repeated again to deal with the execution structures that may remain in the graph due to the nested structure of the activity diagrams.

The PROGRES program described in this chapter was implemented and thoroughly tested. Its application to two case studies is described in chapter 6.

5.5 How long it takes to execute the reduction algorithm

The most time-consuming operations in our reduction algorithm are automatically executed by PROGRES, and are not under the control of the user:

- a) Finding a subgraph in the host graph that matches the left-hand-side of a production rule.
- b) Database access: PROGRES uses an object-oriented database for storing the production rules, so database operations introduce a considerable overhead.

c) **Graphical display:** the tool is interactive and the results are displayed graphically.

We expect that the algorithm will not scale up linearly with the size of the graph, but we did not try it with large enough graphs to experience this effect. For example, the reduction of the largest case study, which has about one hundred nodes, was completed in about 9 seconds on a Sun Ultra 10 workstation (under Solaris 2.7 operating system, with an UltraSPARC-III, CPU of 440 MHz, and 128-Mbyte DRAM). The total execution time included the work undertaken by the PROGRES behind the scenes and the delay for display.

Chapter 6 Case Study

This chapter shows the application of the first SPE step based on activity diagrams to three case studies. Two of these case studies represent the analysis of two systems, a Computer Aided Design system and an ATM banking system. The third example represents a test case for an activity diagram with multiple branches.

6.1 Computer Aided Design Case Study

In this section, we continue the case study from chapter 3. First, we will show how the authors Smith and Williams used a combination of extended sequence diagrams and execution graphs to analyze the best-case response time. Next, we show how to use activity diagrams and our reduction algorithm for the same purpose.

6.1.1 Using Message Sequence Charts with Execution Graphs

As discussed in Chapter 3, Smith and Williams presented and illustrated the SPE process for object-oriented software architecture evaluations in [Williame+98]. They used Message Sequence Charts (MSC) [ITUT-99], which they named “extended sequence diagrams”, to represent alternate scenarios. The MSC models were translated by hand into execution graphs, and their performance was analyzed with the tool SPE.ED^{TP}. In fact, the notation used in [Williams+98] is not “pure” UML.

This case study examines an interactive system, known as ICAD, to support computer-aided design (CAD) activities. Engineers can use the ICAD to construct and view drawings that model structures, such as aircraft wings. The purpose of this case study is to illustrate the use of the SPE to evaluate application architecture and to demonstrate the importance of inter-component communications in determining performance.

Figure 6.1 shows the high-level view of the DRAWMOD scenario for the ICAD system.

Figure 6.1 The DrawMod Scenario

There are three different architectures for the ICAD.

- **Architecture 1**

This architecture uses objects to represent each beam and node. It offers a great deal of flexibility, makes it possible to treat all types of elements in a uniform way, and

allows the addition of new types of elements without the need to change any other aspect of the application. Figure 6.2 shows a more detailed sequence diagram for DRAWMOD in the case of architecture 1. The SPE.ED™ performance engineering tool [Smith90][Williams+98] is used to compute the model solutions.

- **Architecture 2**

Although architecture 1 provides a great deal of flexibility, using an object for each node and beam is potentially expensive in terms of both run-time overhead and memory utilization. To reduce this overhead, the Flyweight pattern is used [Gamma+95]. Using the pattern in ICAD allows the sharing of beam and node objects, and reduces the number of objects that must be created in order to display the model. Each model now has exactly one beam and node object. The node and beam objects contain “intrinsic” states, the information that is independent of a particular beam or node (such as coordinates). They also know how to draw themselves. “Extrinsic” states, coordinates and other information needed to store the model are stored separately. This information is passed to the beam and node flyweights when it is needed. Figure 6.3 shows the sequence diagram for DRAWMOD using architecture 2.

- **Architecture 3**

This architecture is similar to architecture 2. However, it modifies the database management system with a new operation to retrieve a block of data with one call: retrieve Block(). Architecture 3 uses this new operation to retrieve the beams and nodes once at the beginning of the scenario, and stores the data values for all beams and nodes with the model object rather than retrieve the value from the database each

time it is needed as in Architecture 1 and 2. Figure 6.4 shows a sequence diagram for DRAWMOD using architecture 3.

As shown in chapter 3, the response time requirement for the system is 10 seconds. The authors of [Williams+98] have used the SPE.ED™ tool to compute the best-case response time for the three architectures. Table 6.1 lists the results.

Figure 6.2 DrawMod Use Case scenario for Architecture 1

Figure 6.3 DrawMod Use Case scenario for Architecture 2

Figure 6.4 DrawMod Use Case scenario for Architecture 3

Number of Architectures \ Response Time (second) (no Contention)	SPE.ED ^{TD} (Performance Engineering Tool)
Architecture 1	992.329
Architecture 2	992.267
Architecture 3	7.964

Table 6.1 The Response time for each Architecture by SPE.ED^{TD} tool

From the above table, we observe that the first and second architectures do not meet by far the performance requirement that is less 10 second. Only the third architecture can meet the performance requirement. Its substantial improvement is due to a more careful use of the database, by eliminating the huge number of database accesses made in the previous solutions. So architecture 3 will be selected as the final design for the software system. It is important to note that this architecture may still have performance problems in a case when multiple users are contending for the system. However, such analysis requires the user to map the software execution model to a performance model, which is beyond the scope of this thesis.

This case study presents and illustrates the SPE process for software architecture evaluations. Different architectural solutions can include different system resources, and make different resource usage. This shows that the software architecture determines the quality of a software system. While a good architecture cannot guarantee attainment of quality goals, a poor architecture can prevent their achievement. The assessment of the performance characteristics of software architectures is discussed in early life cycle

stages by using estimates for execution times. Errors made in early design stages may cause excessive overhead for communication and coordination, and they are far more difficult, if not impossible, to correct with code tuning. The example shows that the most of the problems were due to a very large number of I/O operations executed by database functions. Even in very early stages it is possible to identify major performance problems such as these, and consider new design alternatives to avoid excessive overhead.

6.1.2 Using Activity Diagram to Analyze Performance

As noted before, in this thesis we propose to use activity diagrams for representing the scenarios in the first step of SPE. However, some object-oriented designers may prefer to use sequence diagrams for representing scenarios. The SPE solution we propose can accommodate this preference. This is work done by other students [Petriu+00b] [Amer01], which automatically transforms sequence diagrams into equivalent activity diagrams.

This section uses activity diagrams that were created by hand for analyzing the three architectures discussed in the previous section.

6.1.2.1 The Activity Diagrams for Three Architectures

- **Architecture 1**

In Figure 6.2, the architecture was described with a UML sequence diagram. Now, we represent the same architecture with a UML activity diagram in Figure 6.5. In separate swimlanes we represent the concurrent components of the system, in order to show their responsibilities with respect to different activities. The data flow between the

concurrent components is represented as `ObjectFlowState`. As explained in section 6.1.1, the sending of a messages is represented as a fork, and the receiving of a message as a join. The `ObjectFlowStates` are treated by the reduction algorithm as all the other States.

Figure 6.5 Activity Diagram for Architecture 1 of DrawMode Scenario 111

Figure 6.5

DrawMode Scenario for Architecture 1 (Continued)

Figure 6.5 DrawMode Scenario for Architecture 1 (Continued)

● Architecture 2

The architecture 2 was described with a UML sequence diagram in the Figure 6.3. Now, we use UML activity diagram to represent the same architecture in Figure 6.6.

Figure 6.6 Activity Diagram for Architecture 2 of DrawMode Scenario

Figure 6.6

DrawMode Scenario for Architecture 2 (Continued)

Figure 6.6 DrawMode Scenario for Architecture 2 (Continued)

● Architecture 3

In Figure 6.4, the architecture 3 was described with a UML sequence diagram. Now, we represent the same architecture with UML activity diagram in Figure 6.7.

Figure 6.7 Activity Diagram for Architecture 3 of DrawMode Scenario

Figure 6.7 Activity Diagram for Architecture 3 of DrawMode Scenario (Continued) 118

6.1.2.2 The Results for the Case Study

Appendix A shows some parts of the PROGRES code for this case study. Table 6.2 shows the best-case response time. The results for the three architectures obtained with the PROGRES tool are almost identical to those published in [Williams+98].

Number of Architectures \ Response Time (second) (no Contention)	PROGRES (Performance Engineering Tool)
Architecture 1	992.516
Architecture 2	992.298
Architecture 3	7.967

Table 6.2 The best-case response time for each Architecture

6.2 ATM Banking System Case Study

This section describes a Banking System Case Study that is also a client /server case study. This case study has been used to illustrate the use of composite activity diagrams in our methodology for reducing activity diagrams.

6.2.1 General Description of the Case

In the Banking System, which is similar to an example from [Gommaa00], a bank has several automated teller machines (ATMs), which are geographically distributed and connected via a wide area network to a central server. Each ATM machine has a card

reader, a cash dispenser, a keyboard/display, and a receipt printer. By using the ATM machine, a customer can withdraw cash from either a checking or savings account, query the balance of an account, or transfer funds from one account to another. A transaction is initiated when a customer inserts an ATM card into the card reader. Encoded on the magnetic strip on the back of the ATM card are the card number, the start date, and the expiration date. Assuming the card is recognized, the system validates the ATM card to determine that the expiration date has not passed, that the user-entered PIN (personal identification number) matches the PIN maintained by the system, and that the card is not lost or stolen. The customer is allowed three attempts to enter the correct PIN; the card is confiscated if the third attempt fails. Cards that have been reported lost or stolen are also confiscated.

If the PIN is validated satisfactorily, the customer is prompted for a withdrawal, query, or transfer transaction. Before a withdrawal transaction can be approved, the system determines that sufficient funds exist in the requested account, that the maximum daily limit will not be exceeded, and that there are sufficient funds at the local cash dispenser. If the transaction is approved, the requested amount of cash is dispensed, a receipt is printed containing information about the transaction, and the card is ejected. Before a transfer transaction can be approved, the system determines that the customer has at least two accounts and that there are sufficient funds in the account to be debited. For approved query and transfer requests, a receipt is printed and the card ejected. A customer may cancel a transaction at any time; the transaction is terminated and the card is ejected. Customer records, account records, and debit card records are all maintained at the server.

Figure 6.8 shows the Use Case Model for the Banking System.

Figure 6.8 Use Case Model for Banking System

6.2.2 Scenarios Modeled by Activity Diagrams

We use activity diagrams with composite activities to represent the main scenarios corresponding to the use case model from Figure 6.8. Figure 6.9 shows a high-level activity diagram that combines all the four use cases. This diagram contains four composite activities, Validate PIN, Withdraw Funds, Query Account, and Transfer Funds. Each one of these composite activities contains inside a detailed activity diagram, which are shown in the Figures 6.10 to 6.13.

In order to obtain the best-case response time, we can use the same methodology to reduce the activity diagrams. First, the lower level activity diagrams are reduced, such as Validate PIN, Withdraw funds, Query account, and Transfer funds. Next the higher level activity diagram is reduced until no unreduced activity diagram is left. The same graph schema, transformation rules, and control algorithms as presented in CAD case study can be used for the ATM banking system case.

- **Higher level activity diagram** is given in Figure 6.9, which contains composite activities that are detailed in the lower-level activity diagrams.
- **Lower level activity diagrams.** Four subactivity diagrams are shown in the following figures: Validate PIN in Figure 6.10, Withdraw funds in Figure 6.11, Query account in Figure 6.12, and Transfer funds in Figure 6.13.

Figure 6.9 High-level Activity Diagram for ATM Case Study

Figure 6.10 Activity Diagram for Validate PIN in ATM case study

Figure 6.11 Activity Diagram for Withdraw Funds in ATM case study

Figure 6.11 Activity Diagram for Withdraw Funds of ATM case study (Continued)

Figure 6.12 Activity Diagram for Query Account in ATM case study

Figure 6.13 Activity Diagram for Transfer Funds in ATM case study

6.2.3 Verification

We may use the same approach presented in the CAD case study to get the best-case response time for the case study. However, we wanted to simplify the testing of the case study, so instead of using more realistic execution times, we assume that the execution time for each node equals 1 second. The results obtained for the low level and high level activity diagrams are given in Table 6.3. The results match those obtained by hand and show that the algorithm behaves correctly.

Name of Activity Diagram	Execution Time (second)
Validate PIN	8
Withdraw Funds	13
Query Account	9
Transfer Funds	8
Total execution time	45

Table 6.3 The Execution Time for ATM Banking System Case Study

6.3 Testing Case for Multiple Branches

Different test cases were used for different activity diagram structures. In this section we present only one of them, which tests the reduction of a six-branch structure as shown in Figure 6.10.

The PROGRES code used to generate the input graph for this activity diagram is given in Appendix B. The code used for reduction is the same for all case studies and does not depend on the particular structure of the input graph.

Figure 6.11 shows a screen shot of the input graph for this test case, as displayed by the PROGRES tool. Figure 6.12 shows the reduced graph and the best-case execution time result as displayed by the tool (which is the same as the result computed by hand)

As mentioned in section 5.3.3, when the transaction for reducing a branching structure is executed, the branches are reduced in a non-deterministic order. However, the final result of the reduction will be the same, as we do not mix the reduction of different structures. The experiments have shown that, the PROGRES behaves, indeed, nondeterministically. No matter how many times we executed the test, the final result was always the same.

Figure 6.14 Six Branches' Structure and combination other Structures

Testing Result

In this section, we show three figures which are an input graph for the testing case, an output graph, and with a contained total execution time as an attribute.

Figure 6.15 Input Graph for the Multiple Branches Testing Case

Figure 6.16 The Final Graph and Execution Time for the Case

From Figure 6.16, we see that the response time for this case is 14 seconds, exactly the same as the result obtained "by hand".

In the chapter, we demonstrated three case studies to verify our methodology. We show how to use our proposed method to obtain performance characteristics to analyze SPE. Next chapter, we will give conclusions which summarize indicate further researches.

Chapter 7 Conclusions

This chapter summarizes the work performed in this thesis and presents the main areas which require additional research and work to be done.

7.1 Conclusions

In this thesis, we have proposed to perform the first step of the Software Performance Engineering (SPE) methodology introduced in [Smith90] entirely in the UML domain. For this purpose, we have proposed to use activity diagrams for representing the SPE Software Execution Model. A reduction algorithm for activity diagrams based on graph transformations was proposed, implemented and tested. We have used an existing graph rewriting tool named PROGRES [Schüerr97c] for the implementation. The activity diagram reduction algorithm was applied to two case studies and many test cases. The work of the thesis is a part of a larger effort to transform UML models into performance models.

During the thesis, we reached the following conclusions regarding the use of graph transformations:

- The graph transformation approach works for the proposed application, although it is not the only approach. For example, one could program in Java or C++ the algorithm for computing the aggregate time without actually using a graph rewriting tool or even without reducing the graph to a single node. However, this solution of writing an ad-hoc program does not easily solve the larger problem of translating a UML model into a performance model.
- For people who are not familiar with graph grammars, there is a very steep learning curve needed in order to understand and successfully use a graph-grammar tool like PROGRES.
- PROGRES is quite slow and introduces a lot of execution overhead due to the fact that the production rules are stored in a database, and many database accesses are performed behind the scenes.
- Interconnecting PROGRES with a UML tool is possible by translating into a PROGRES input file the XML file containing the UML model.

7.2 Future Work

A short-term goal for this research is to extend the reduction algorithm in order to reduce an activity diagram by swimlane. This way, the resource demand vector can be computed for each partition.

A longer-term research goal of the research is to integrate the activity diagram reduction in the larger project that transforms UML models into LQN performance models. The final product will be a graph transformation based tool that will accept as

input a UML model and will produce as output an LQN model. Such a tool will be useful for automating the SPE steps that require the construction of a performance model. Another direction of research would be to interconnect a UML tool with our PROGRES program. The UML tools can produce XML files containing the UML model (according to the standardized XMI interface [UML 1.3]). An XML parser could be use to read and interpret such a model, then to produce a host graph in the format understood by PROGRES.

References

- [Amer01] Amer,Hoda "Automatic Transformation of UML Software Specification Into LQN Performance Models Using Graph Grammar Techniques" Master's Thesis, Carleton University, Department of Systems and Computer Eng., May 2001.
- [Berge58] Berge,C., "Theorie des Graphes et Ses Applications" Paris,Dunod,1958.
- [Booch+99] Booch, B. and Rumbaugh, J. and Jacobson, I. "The Unified Modeling Language User Guide", Addison-Wesley, 1999.
- [Bochmann00] Bochmann, Gregor V. "Activity Nets: A UML profile for modeling workflow and business processes" .
- [Bezivin+98] Bezivin, Jean and Muller, Pierre-Alain " The Unified Modeling Language" <<UML>>' 98: Beyond the Notation: first international workshop.
- [Booch+98] Booch, G., J. Rumbaugh, and I. Jacobson. "The Unified Modeling Language User Guide" Reading, Mass. : Addison-Wesley, 1998.
- [Burns+96] Burns, A., and Wellings, A. J., : "Real-Time Systems and their Programming Languages" 2nd, ed. Reading Mass.: Addison-Wesley, 1996.
- [Bass+98] Bass,L.,P, Clements and R. Kazman " Software Architecture in Practice " Reading Mass.: Addison-Wesley, 1998.
- [Chichester95] Chichester, England, Wiley. "Modelling with generalized stochastic Petri nets." 1995 301 p.
- [Cortellessa+00] Cortellessa,V. and Mirandola, R. "Deriving a Queueing Network based Performance Model from UML Diagrams", Proceedings of the Second International Workshop on Software and Performance, Ottawa, Sept. 2000.
- [Douglass98] Douglass, Bruce Powel "Real-time UML" Addison-Wesley, 1998.
- [Douglass+99] "Doing Hard Time Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns" Addison-Wesley. 1999.
- [Eriksson+98] Hans-Erik, Eriksson and Penker, Magnus "UML Toolkit" New York : John Wiley & Sons, 1998.
- [Evans+00] Evans, Andy and Kent, Stuart and Selic, Bran " <<UML 2000>> The Unified Modeling Language Advancing the Standard" Third International Conference York, UK, October 2000 Proceedings.
- [Fahmy+00] Fahmy, H. And Holt, R.C. "Software Architecture Transformations", to appear in the Proceedings of the International Conference on Software Maintenance, San Jose. Oct. 2000.
- [Fenton+97] Fenton, N.E. and Pfleeger, S. L. "Software Metrics: A Rigorous and Practical Approach" PWS Publishing Company, 1997.
- [France+99] France, Robert and Rumpe, Bernhard <<UML>>' 99- "The Unified Modeling Language" Beyond the Standard. 2nd international conference.
- [Franks99] Franks, G., Performance Analysis of Distributed Server Systems, Ph.D Thesis, Department of Systems and Computer Engineering, Carleton University , 1999.

- [Fowler+99] Fowler, Martin "UML Distilled Applying the Standard Object Modeling Language" Addison-Wesley, 1999.
- [Gamma+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software" Reading MA, Addison-wesley, 1995.
- [German+00] German, Reinhard and Chichester, England, Wiley "Performance analysis of communication systems; modeling with non-Markovian stochastic Petri nets", 2000 438 p.
- [Gomaa00] Gomaa, Hassan " Designing Concurrent, Distributed, and Real-Time Applicatons with UML" ADDISON-WESLEY Pearson Education, 2000.
- [Hoffmann+] Hoffmann, Berthold and Krieg-Bruckner, Bernd "Program Development by Specification and Transformation" The PROSPECTRA methodology, Language Family, and System.
- [Hrischuk+99] Hrischuk, Curtis. E. and Woodside, C. Murray and Rolia, Jerome A. and Iversen, Rod "Trace-based load characterization for generating performance software models" IEEE Transactions on Software Engineering, Vol.25, Nb. 1, pp 122-135, January 1999.
- [Hrischuk+95] Hrischuk, C., Rolia, J., and Woodside, C. M. "Automatic Generation of a Software Performance Model Using an Object-oriented Prototype", Proceedings of the third Internatinal Workshop on Modeling, Analysis, and Simulation of computer and Telecommunication systems, Durham, NC, January, 1995, pp 399-409.
- [Hrischuk98] Hrischuk, Curtis E. "Trace-Based Load Characterization for The Automated development of Software Performance Models" Ph.D. Thesis, Carleton University Department of Systems and Computer Eng., May 1998.
- [ITUT-99] International telecommunication Union (ITU-T Z. 120), Messages Sequence chart (MSC), 11/99.
- [Jacobson+99] Jacobson, I., G. Booch, and J. Rumbaugh. "The Unified Software Development Process" Reading, Mass. : Addison-Wesley, 1999.
- [Kiesel+93] Kiesel, N. and Schuerr, A. and Westfechtel, B. and GRAS, "A graph-oriented database system for (software) engineering applications." In H.-Y. Lee, T. F. Reid, and S. Jarzabek, editors, Proceedings of the Sixth international Workshop on Computer-Aided Software Engineering CASE 93, pages 272-286 1993.
- [Larman] Larman, Craig "Applying UML and Patterns" Upper Saddle River, N.J., Prentice Hall PTR 1998 507p.
- [Marshall] Marshall, Chris "Enterprise Modeling with UML".
- [Molloy] Molloy, Michael K. "Fundamentals of Performance Modeling".
- [Oestereich] Oestereich, Bernd "Developing Software with UML" Object-Oriented Analysis and Design in Practice.
- [OMG01] "Response to the OMG RFC for Schedulability, Performance and Time", Version 1.8, June 14, 2001.
- [OMG] OMG, editor. Meta Object Facility (MOF) Specification, Version 1.3 RTF, 2 July 1999. Object Manaagement Group, Inc., Framingham, Mass., Internet: <http://www.omg.org>, 1999.

- [Page-Jones] Page-Jones, Meilir. "Fundamentals of Object-oriented Design in UML" New York, Dorset House Publishing. 2000.
- [Petriu01] Petriu, Dorina C., "Deriving Performance Models from UML specifications by Graph Transformations" Presentation in IEEE in Ottawa, 2001.
- [Petriu 00a] Petriu, Dorina C. "Deriving Performance Models from UML Models by GraphTransformations", proceedings of the Second International Workshop on Software and Performance, Ottawa, Sept. 2000.
- [Petriu+00b] Petriu, D. C. and Sun, Y. "Consistent Behaviour Representation in Activity and Sequence Diagrams", to appear in Proc. Of UML'2000, York, GB, October 2000.
- [Petriu+00c] Petriu, D. C. and Jalnapurkar, C.shousha,A. "Architecture-Based Performance Analysis Applied to a Telecommunication System", to appear in IEEE Transactions on Software Eng, Vol.26, No.11, pp.1049-1065, Nov. 2000.
- [Petriu +99] Petriu, D. C. And Wang, X. "From UML Descriptions of High-Level Software Architectures to LQN Performance Models" Proc. Of Applications of Graph Transformation with Industrial Relevance, Monastery Rolduc, Kerkrade, The Netherlands, September 1999.
- [Petriu+98] Petriu, D. C. and Wang, X. "Deriving software Performance Models form Architectural Patterns by Graph Transformations", in Theory and Applications of Graph Transformations, TAGT'98 (H.Ehrig, G.Engels, H.J. Kreowski, G. Rozenberg, Eds.) Lecture Notes in Computer Science 1764, pp475-488, Springer Verlag, 1998.
- [Pooley+99] Pooley,T., Stevens, P. "Using UML: Software Engineering with Objects and Components" Addison Wesley Longman, 1999.
- [Rumbaugh+99] Rumbaugh, J., Booch, G., Jacobson, I. "The Unified Modeling Language Reference Manual" Addison Wesley Longman, 1999.
- [Schüerr90] Schüerr,A., "Introduction to PROGRES, an attribute graph grammar based specification language", in Graph-theoretic Concepts in Computer Science, M. Nagl, editor, Vol 411 of Lecture Notes in Computer Science, pp 151-165, 1990.
- [Schüerr94] Schüerr, A., "PROGRES: A Visual Language and Environment for PROgramming with Graph Rewrite Systems, Technical Report AIB 94-11, RWTH Aachen, Germany, 1994.
- [Schüerr97a] Schüerr, A., "Programmed Graph Replacement Systems" in Handbook of Graph Grammars and Computing by Graph Transformation, G. Rozenberg(ed), pp. 479-546, 1997.
- [Schüerr97b] Schüerr, A., "A Guided Tour through the PROGRES Environment" RWTH Aachen, D-52056 Aachen, Germany.
- [Schüerr97c] Schüerr, A., "PROGRES for Beginners" RWTH Aachen, D-52056 Aachen, Germany. 1997
- [Shaw96] Shaw, M. "Some Patterns for Software Architecture" in Pattern languages of Program Design 2 (J. Vlissides, J. Coplien, and N.Kerth eds.), pp.255-269, Addison Wesley, 1996.
- [Smith90] Smith, C.U., "Performance Engineering of Software Systems", Addison

- Wesley, 1990.
- [Smith+98] Smith, C.U. and Williams, L.G., "Software Performance Engineering for Object-oriented Systems: A Use Case Approach" submitted for publication, 1998.
- [Smith+97] Smith, C.U. and Williams, L.G., "Performance Engineering Evaluation of Object-oriented systems with SPE.ED" in Computer Performance Evaluation: Modelling Techniques and Tools, Lecture Notes in Computer Science, vol. 1245, R. Marie, B Plateau, M. Calzarossa and G. Rubino, ed., Berlin, Springer-Verlag, 1997, pp 135-154.
- [Smith+93] Smith, C.U. and Williams, L.G., "Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives", IEEE Transactions on Software engineering, vol. 19, no. 7, pp. 720-741, 1993.
- [Sventek+00] Sventek, Joseph and Coulson, Geoffrey "Middleware 2000 "IFIP/ACM International conference on Distributed Systems Platforms New York, USA, April 2000 Proceedings.
- [Sinan98] Sinan, Si Alhir "UML In A Nutshell" Paula Carroll 1998.
- [UML 1.3] OMG: Unified Modeling Language Specification, Version 1.3, 1999.
- [UML 1.4] OMG: Unified Modeling Language Specification, Version 1.4, 2001.
- [Wang99] Xin Wang "Deriving Software Performance Models From Architectural Patterns By Graph Transformations" Master's Thesis.
- [Williams+98] Williams, L.G, Smith, C. U., "Performance Evaluation of software Architectures", Proceedings of the First International Workshop on software and Performance, Santa Fe, USA, pp. 164-177, Oct. 1998.
- [Williams+95] Williams, L. G. and Smith, C. U. "Information Requirements for Software Performance Engineering", in quantitative Evaluation of Computing and Communication Systems, Lecture Nodes in Computer Science, Vol. 977, H. Beilner and F. Bause, ed., Heidelberg, Germany, springer-Verlag, 1995, pp 86-101.
- [Winston+87] Winston, M.E. and Chaffin, R. And Hermann, D. "A Taxonomy of Whole-part Relations" Cognitive Science, Vol. 11, 1987.
- [Woodside89] Woodside,C. M., "Throughput Calculation for Basic Stochastic Rendezvous Networks", Performance Evaluation , Vol. 9, Number 2, pp. 143-160, April 1989.
- [Woodside+95] Woodside, C.M. Neilson, J. E. Petriu, D.C Majumdar, S., "The Stochastic Rendzvous Network Model for Performance of Synchronous Client-Server-like Distributed Software", IEEE Transactions on Computers, Vol. 44, Nb. 1, pp. 20-34, January 1995.

Appendix A: Parts of the PROGRES Program for the ICAD Case Study

- a) The main control structure of the case.

```
transaction    MAIN =  
    CreateMixPath  
  
    when def ( StillHaveTwoNodesLeft )  
    then  
  
        & loop  
            AggregationAllSequenceNodes  
            & AggregationAllMoreThanTwoBranches  
            & AggregationAllCasePathNodes  
            & AggregationAllLoopPathNodes  
            & AggregationAllParallelPathNodes  
        end  
    end;  
  
end.
```

- b) Code to generate the input graph (due to space limitations, we omit the most of code.

```
section Test  
  
transaction CreateMixPath =  
    (<OptPrecondDecl>  
        CreateDiagram  
        ( "NodeBegin", "Trans_1", "State_1", "Act_1",  
            "Trans_2", "State_2", "Act_2", "Trans_3", "NodeEnd")  
        & InsertNode  
        ....
```

end;

c) Code for Transaction AggregationAllSequenceNodes

```
transaction AggregationAllSequenceNodes =  
  (<OptPrecondDecl>)  
    loop  
      AggregatSequenceNodes  
    end  
end;
```

```
transaction AggregationAllCasePathNodes =  
  loop  
    AggregatCasePathNodes  
  end  
end;
```

```
transaction AggregationAllLoopPathNodes =  
  loop  
    AggregatLoopPathNodes  
  end  
end;
```

```
transaction AggregationAllMoreThanTwoBranches =  
  loop  
    AggregatMoreThanTwoBranches  
  end  
end;
```

```
transaction AggregationAllParallelPathNodes =  
  loop  
    AggregatParallelPathNodes  
  end  
end;
```

Appendix B: PROGRES Code for Multiple Branches Test

Code for creating the input graph

section Test

```
transaction CreateMixPath =  
(<OptPrecondDecl>  
    CreateDiagram( "NodeBegin", "Trans_1", "State_1",  
        "Act_1", "Trans_2", "State_2", "Act_2",  
        "Trans_3", "NodeEnd" )  
    & InsertNode("Trans_2", "State_8", "Act_8", "Trans_8",  
        "State_2", 1 )  
    & InsertParallelPathNodes( "Trans_2", "ForkNode_1",  
        "Trans_14", "Trans_15", "State_14", "Act_14",  
        "State_15", "Act_15", "Trans_16", "Trans_17",  
        "JoinNode_1", "Trans_18", "State_8", 2, 1, 4 )  
    & InsertNode( "Trans_8", "State_9", "Act_9", "Trans_9",  
        "State_2", 1 )  
    & InsertSixBranches( "Trans_8", "Branch_1", "Guard_3",  
        "Trans_3", "Trans_4", "Guard_4", "Trans_10",  
        "Guard_10", "Trans_11", "Guard_11", "Trans_12",  
        "Guard_12", "Trans_13", "Guard_13", "Act_3",  
        "State_3", "State_4", "Act_4", "State_10",  
        "Act_10", "State_11", "Act_11", "State_12",
```

```

    "Act_12", "State_13", "Act_13", "Trans_3b",

    "Trans_4b", "Trans_10b", "Trans_11b", "Trans_12b",

    "Trans_13b", "Merge_1", "Trans_Merge", "State_9",

    6, 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6 )
& InsertNodeBetweenStateAndTransitionOnly
    ( "State_4", "Trans_5", "State_5", "Act_5",

    "Trans_4b", 1 )
& InsertLoopPathNodes( "Trans_9", "LoopBegin_1",

    "Trans_6a", "Guard_6a", "State_6", "Act_6",

    "Trans_6b", "LoopEnd_1", "Trans_6c", "State_2",

    1, 2 )
& InsertNodeBetweenTransitionAndLoopEnd
    ( "Trans_6b", "State_7", "Act_7", "Trans_7",

    "LoopEnd_1", 1)
& EnterAttributeForArcNode
    ( "Guard_3", "GuardNode", "BranchNode_1",

    "State_3", 1 )
& EnterAttributeForArcNode
    ( "Guard_4", "GuardNode", "BranchNode_1",

    "State_4", 1 )
& EnterAttributeForArcNode
    ( "Guard_10", "GuardNode", "BranchNode_1",

    "State_10", 1 )
& EnterAttributeForArcNode
    ( "Guard_11", "GuardNode", "BranchNode_1",

```



```

        "State_11", 1 )
    & EnterAttributeForArcNode
        ( "Guard_12", "GuardNode", "BranchNode_1",

        "State_12", 1 )
    & EnterAttributeForArcNode
        ( "Guard_13", "GuardNode", "BranchNode_1",
        "State_13", 1 )
    & EnterExecutTime ( "Act_1", 1 )
    & EnterExecutTime ( "Act_2", 1 )
end;

```