System Family Engineering for Leader Election in Ring

by

Wei Shi

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master's of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

8th December 2003

Copyright ©2003 by Wei SHI

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisitions et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-89864-4

Our file Notre référence

ISBN: 0-612-89864-4

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.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this dissertation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de ce manuscrit.

While these forms may be included in the document page count, their removal does not represent any loss of content from the dissertation.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

In this thesis we present an executable and testable model for a family of algorithms dealing with Leader Election in a Ring topology. We first follow the traditional approach of System Family Engineering. That is, we develop a Feature Model that captures variability across these algorithms. Beyond the Feature Model, we then proceed to produce a test-ready executable State Machine model. This state machine model receives specific values for each of the variation points (i.e., features) we identify as inputs. And it produces the behavior corresponding to the specific configuration of features at hand. Contrary to existing generative programming literature, we do not resort to advanced C++ programming, but instead develop an executable model using Rational Rose RT. More precisely, we have designed an integrated State Chart that can execute all the algorithms of the family we studied. We do believe that our approach can be reused to provide a semantically unified, test-ready and executable modeling approach for other families of algorithms.

Acknowledgements

I would like to give my sincere thanks to my main supervisor, professor Jean-Pierre Corriveau for suggesting the topic, then teaching us about generative modeling and programming. He has devoted so much time and effort to teaching me both in this research and in writing it that my labors will never be able to match it. His vision and his dedication made this dissertation a reality.

I would also like to thank professor Nicola Santoro for his expertise in distributed algorithms, which made this thesis possible.

Special thanks to Li Zhao, David Arnold, and Juan Pablo Zamora Zappata for all the help with ROSE-RT.

Many thanks as well to Hua Guo, Xinrong Huang, Xingyuan Liu, Zheng Li, Thierry Metais, An Sheng, and also Qi Deng, Boyong Liang. As friends and classmates, their help and support gave me great courage and confidence to keep working

Finally, I would like to express my true appreciation for my parents. If it was not their constant support and great encouragement and endless love, my academic life to achieve this degree would have been much longer.

Contents

1	Introduction	1
1.1	The Context	1
1.1.1	The Motivation	1
1.1.2	System Family Engineering	4
1.2	Some Terminology	8
1.3	The Method	11
1.4	Contributions	13
1.5	Plan for the Thesis	15
2	The Domain: Leader Election in a Ring	16
2.1	More Terminology for This Chapter	16
2.2	The ring election problem	19
2.2.1	Leader Election	19
2.2.2	Restrictions	21
2.2.3	Leader Election in Rings	24

2.3	The selection of algorithms	25
2.4	Initial Conceptual Considerations	30
3	An Integrated Leader Election Algorithm	33
3.1	Functional requirements and Non-functional Requirements	33
3.1.1	Functional requirements	33
3.1.2	Non-functional requirements	34
3.2	Initial feature modeling	35
3.2.1	Choosing Features	35
3.2.2	Distinguishing Features and Parameters	36
3.2.3	Selection of Features and Parameters	36
3.3	Details of Features	38
3.3.1	Features and Feature Values Definition	38
3.3.2	Feature Interactions - Combination Rules	49
4	Beyond Feature Model	53
4.1	State Machine Model	53
4.1.1	Why State Machine?	53
4.1.2	What is State Machine?	54
4.2	An Intermediate Model to Fill the Gap between Feature Model and State Machine Model	58
4.2.1	Motivation	58

4.2.2	Decision Trees and Flow Charts	58
4.3	Feature-State-Decision-Flow Chart	60
5	An Integrated Executable Model	63
5.1	Towards An Integrated Executable Model	63
5.2	A Crash Course in ROSE-RT	66
5.3	Two Individual State Charts for Two Algorithms.	69
5.4	Integrating the first two algorithms in Rational Rose RT.	76
5.5	Integrating "Single Stage" and "Multiple Stage"	83
6	Conclusion	92
6.1	About our Contributions	92
6.2	Future research in the context of this thesis	94

List of Tables

2.1	Feature Algorithms relationship.	31
3.1	Identifiers for feature values	39

List of Figures

1.1	Feature Model for Car Example	9
2.1	The Example for Protocol "Stages with Feedback"	29
3.1	Top Level Feature Diagram for Leader Election in Ring	40
3.2	Oriented Ring	41
3.3	Unoriented Ring	42
3.4	Feature Diagram for Leader Election in Ring–Topology Group . . .	43
3.5	Feature Diagram for Leader Election in Ring–Protocol Group	47
3.6	Feature Diagram for Leader Election in Ring–Node Behavior Group	49
3.7	Feature Diagram for Leader Election in Ring Topology Domain . .	50
4.1	State Transition Diagram example	57
4.2	Decision Tree example	59
4.3	Flow Chart Example	61
4.4	FSDF Chart Notation	62
5.1	Class Diagram with Capsules	70

5.2	The State Chart for Protocol "Asfar" (Chang and Robert's algorithm)	72
5.3	The State Chart for Protocol "All the way" (LeLann's algorithm) . .	77
5.4	The Feature value - Algorithm Corresponding diagram for protocol "All the way" and protocol "AsFar")	78
5.5	The integrated state machine for two algorithms - Event code on transition from state Candidate to state Terminate	81
5.6	The integrated state machine for two algorithms - Action code on transition from state Candidate to state Terminate	82
5.7	The integrated state machine for two algorithms - Action code on self transition on state Candidate	83
5.8	The integrated state machine for two algorithms - Action code on self transition on state Candidate to state Terminate	84
5.9	The integrated state machine – Action code on self transition on state Candidate to state Terminate	86
5.10	The integrated FSDF Chart	87
5.11	Capturing Feature Combination Rules in the top level state diagram in integrated model	88
5.12	Second level state diagram of integrated executable model	90
5.13	A Running State Machine Model on One of the Node in the Ring . .	91

Chapter 1

Introduction

1.1 The Context

1.1.1 The Motivation

A current trend in software engineering is reuse: reuse of analysis, reuse of design, reuse of code, and reuse of applications. Software reuse is driven by the desire to reduce software development costs and to leverage existing high quality artifacts. Meanwhile, stakeholders are demanding specialized software that fits their specific needs. Software must therefore be easily customizable.

This creates conflicting forces on software development: reusing existing software artifacts while creating customized high-quality software that meets the specific needs of its stakeholders. Recent developments in software engineering have

attempted to address various aspects of this issue.

Domain driven software development methods, such as software product line engineering, system-family engineering, or generative programming, all promote a generative approach to software production and reuse. More specifically, these approaches address variability in a family of systems. This family forms a domain. Modeling software variability facilitates software reuse: a highly flexible generic architecture and a set of domain components can be reused in order to produce a new software product within the given family. Given that except for small, if not trivial systems, it will generally be impractical to enumerate all possible systems in a family, we need a generative approach to the capturing of variability, one that implicitly specifies family members as valid possible variations. In order to achieve this, the use of feature models is generally advocated. Here, a feature is a "variation point" and a feature model leads to a grammar of valid and invalid combinations of features, and ultimately to a generator (defined as "a program that takes a higher-level specification of a piece of software and produces its implementation".) This generator is then used in the application-engineering process to produce an executable system.

It is important to acknowledge that modifiability is different from variability. Modifiability is the ability for a system to be changeable. It constitutes a non-

functional requirement, often referred to as "ease-of-modification", which can be measured (e.g., with respect to the number of hours required on average to implement a particular type of change in an existing system). Conversely, variability is the ability for a system to vary in its functionality across a domain. For example, a security component offers a variable encryption strategy if this component can use distinct encryption strategies. On the other hand, this component is highly modifiable if it is trivial for a designer to modify the set of encryption strategies this component handles. From this viewpoint, variability pertains to a domain, whereas modifiability is specific to a particular application (i.e., a particular member of a family of applications). Indeed, some family members may optimize modifiability at the expense of performance, whereas others (i.e., other variants in the family) may do exactly the opposite.

Variability and commonality are often de-emphasized in current software development approaches. It is reported that when we compare software systems, we usually find 60% to 70% commonality from one application to a similar one. [42] This includes code, design, functional and architectural similarities. But software reuse (taken as the process of creating software systems from predefined software components [36]) has been shown to cut software development time and costs. This dissertation originates in the commitment of our research group to investigate reuse not just of code, but first and foremost of designs and their decisions from a quality

engineering standpoint [30], [27]. The question at the root of the present work is one from Professor. Corriveau: what about variability and reuse across a group of similar algorithms?

1.1.2 System Family Engineering

Generative modeling and programming is a recent software engineering paradigm based on modeling software system families such that, given a particular set of requirements, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge [10],[37],[32]. More generally, generative programming is about bringing the benefits of reuse to software development in the context of families of systems, as opposed to reuse in single-solution approaches to software development. [13]

It is useful to distinguish between two approaches to generative modeling. For Withey [45] a product-line is *"a group of products sharing a common, managed set of features"* and a system family is *"a group of products that can be built from a common set of assets"*. Product-line engineering applies to the reuse of components and architecture in a family of products. System-family engineering, on the other hand, applies to the development of specialized software that aims at producing,

out of a set of reusable assets, a specific family member customized to a specific configuration of stakeholder requirements. A family of such systems is taken to form a domain. In this dissertation, our focus is on system family engineering. And we add that domain recognition and domain evolution both lie beyond the scope of this work: we start here with an intuitive domain and explore variability within this domain.

Generally, existing approaches to modeling variability rest on a technique called *Feature Modeling* [32],[31]. Throughout this work, we will adopt the viewpoint that a feature is a logical unit of functionality. As previously mentioned, features are variation points across the domain: each one captures a functional facet of the domain that allows designers to distinguish between the different family members. For example, a car having a gasoline or (exclusively) an electric engine establishes two categories of cars (along that feature) in the vast family of cars. Should we remove the exclusive-or and replace it with an inclusive one, with respect to engines, we would partition the domain of cars into three categories (gas, electric, both). From this standpoint, each family member can be thought of as a specific configuration of features, that is, as a specific set of feature values that uniquely distinguishes it from other family members. The fundamental task, then, of a system family (also called domain) engineer is to determine a set of features that adequately partitions family members and to document this set using a feature model. The completeness

and correctness of such a model turn out to be remarkably difficult to measure. For example, completeness would have to be measured using some traceability scheme between requirements, features and code. But, to the best of our knowledge, no one has put forth such an approach. And consequently, we believe it is premature to try to assess the completeness and the correctness of such models. Consequently, we suggest our work be viewed first as a feasibility study: our goal is to demonstrate that the ideas of system family engineering can indeed be applied to a novel domain. As previously mentioned, we will investigate the application of system family engineering to a particular family of distributed algorithms, something that has not been attempted before. We have selected the family of algorithms pertaining to the election of a leader in a ring topology. But before elaborating on this domain per se, we must ask how we should measure success in our work.

Current software reality is that maintenance costs are still astronomical in the software industry [7]. And though reuse was presented as a solution to this problem [29], little evidence of productivity gains has been reported. One probable reason for this is that measuring elusive notions such as reuse, productivity gains, modifiability, and maintainability is remarkably costly and hard. But if we cannot assess the correctness, nor the completeness, nor the potential benefits of the feature models we will develop in this dissertation, how will we know we have achieved our goal of 'adequately' modeling the target domain?

The answer to this question is two-fold. First, as previously mentioned, a good part of the work presented in this dissertation was developed as a feasibility study on the applicability of system family engineering to a family of distributed algorithms. From this viewpoint, success is simple to assess: did we or did we not manage to construct a feature model for the target domain. The short answer is yes. But how to judge the quality of what we put forth in the rest of this dissertation? One objective criterion emerges quickly: executability. Put simply, our thesis is that the feature models we propose here are adequate because we managed to use these features to build an executable and testable model of the domain. More precisely, we have built a single executable model that can be configured (with different sets of feature values) to simulate any of the algorithms that form the domain. Thus, the second facet of assessing our work rests on the testing of this executable model. In our opinion, a significant contribution of our work is that we explain at length how we develop an executable model that is also highly testable. Specifically, our executable model takes the form of a state machine [43] [22] implemented in a commercial modeling tool. And because of that, we can reuse a wealth of strategies (and tools) to extensively test this model [11]. And thus, this does enable us to assert upfront that the proposed feature model and its corresponding executable model indeed 'adequately' address the target domain. It is this demonstration that we develop in the rest of this dissertation.

First, we need to clarify some of the terminology used in system family engineering. We will then introduce the steps of our work and the target domain.

1.2 Some Terminology

In this section, we consider a small example to introduce the terminology used in system family engineering.

Case study – Car

The car consists of a car body, a transmission, and an engine. The transmission may be either manual or automatic. Furthermore, the car may have a gasoline engine or an electric engine or both. Finally, the car may pull a trailer. Figure 1.1 [32] shows a feature diagram of a simple car. Feature Car has three mandatory sub-features: car body, transmission, and engine, and one optional feature value: Pulls trailer. The Transmission has two alternative feature values: Automatic and Manual. The Engine has two or feature values: Electric and Gasoline.

This is a very simple example of feature modeling but we do not require more complicated ones within the context of this work since it allows us to introduce the following definitions:

Figure 1.1: Feature Model for Car Example

Feature diagram:

Represents a hierarchical decomposition of features including the indication of whether or not a feature is mandatory, alternative, or optional.

Feature definitions:

Capture a free form comment on other details of the feature (e.g., responsibilities, binding time, binding location, etc.).

Composition rules for features:

Indicate which feature combinations are valid and which ones are not.

Rationale for features:

Motivates the choice of a particular feature.

Mandatory Feature:

Formally, a mandatory feature is included in the description of a concept instance if and only if its parent is included in the description of that instance. A mandatory feature node is pointed to by a simple edge (as opposed to an arc-decorated edge) ending with a filled circle.

Optional Feature:

An optional feature may be included in the description of a concept instance if and only if its parent is included in the description. In other words, if the parent is included, the optional feature may be included or not, and if the parent is not included, the optional feature can't be included. An optional feature node is pointed to by a simple edge ending with an empty circle.

Alternative Feature: If the parent of a set of alternative features is included in the description of a concept instance, then exactly one feature from this set of alternative features is included in the description, otherwise none are included. The nodes of a set of alternative features are pointed to by edges connected by an arc. A concept or a feature may have one or more sets of direct alternative features.

Or-Feature:

A concept may have one or more sets of direct or-features. Similarly, a feature may have one or more sets of direct or-subfeatures. If the parent of a set of or-features is included in the description of a concept instance, then any nonempty subset from the set of or-feature is included in the description; otherwise, none are included. The nodes of a set of or-features are pointed to by edges connected by a filled arc.

Functional requirements:

Address what the software is supposed to do.

Non-Functional requirement:

Address how the software is supposed to perform. Typical examples include performance, security, safety, modifiability, reliability, usability, etc.

1.3 The Method

In this section, we briefly overview the steps we will follow to explain the construction of a feature model for the target domain and the extraction of an executable model from it.

Step 1: Clarify the Scope of the Domain

Before doing anything else in a family of systems, we need to give a definition of

the domain, that is, to clarify the scope of the domain. In our case, we will review literature on algorithms for leader election in a ring and identify a set of algorithms to consider.

Step 2: Problem Characterization

A significant advantage of our target domain is that it is well scoped and largely researched. Thus we can describe the problem at hand independently of the solutions proposed for it. No domain engineering can proceed unless this is first done.

Step 3: Develop a Feature Model

As previously mentioned, feature modelling is a technique for addressing variability in a domain. Developing a feature model for the domain is the key step in system family engineering. We must investigate features, feature values and combination rules.

Step 4: Develop an Integrated Executable Model for the Domain

Recall that our executable model takes the form of a single hierarchical state machine. From the feature model we propose, we show how to build an intermediate model that allows to move from features to states and sequences of states. With this information we proceed to build the desired executable model in detail. In the tool we have selected, this means detailing in C++ the code associated with each

transition of our executable model.

Step 5: Implement the Executable Model in Rational Rose RT

We select Rational Rose Real Time from IBM to implement our executable model. The advantages of this choice will be discussed later in this dissertation. For now, we emphasize that this choice was mainly motivated by the ease with which a complex state machine can be thoroughly tested using this tool. We also remark that the tool does support distribution and concurrency extensively: there is no simulation code for us to write here; we merely have to build a network and specify the behavior of the node of this network using a single state machine. Then simulation and testing are largely handled by the tool itself!

1.4 Contributions

We believe we make the following contributions to the state of the art:

1. A Feature Model for the Domain of Algorithms for Leader Election in a Ring.

By itself a feature model provides a unified conceptual framework to understand variability across a domain. In the context of the selected family of algorithms, given the apparent diversity, we believe such an integrated view

of the domain will help researchers understand variations and explore new combinations of features.

2. An Approach to "Feature to State" Transition. We do insist that system family engineers so far have produced feature models from which they 'extract' generators. The semantic gap between the models and those complex generators is large and, in practice, feature models are used merely as a first step towards the 'real' deliverable, namely, the generator. Our viewpoint is different. We want a state machine as the target executable model. And we do suggest how to go from features to states.
3. An Executable and Testable Integrated State Machine for the Domain. It is worth repeating that having a single state machine that can be configured to execute any of the algorithms we considered has significant advantages. Not only are the simulation and testing tasks made easy, but we do believe that such a model opens the door to the exploration of novel combinations of features. And we must point out that existing research in system family engineering has completely neglected the issue of testability. From this viewpoint, and given the mandate of our research group, we believe our contribution will be the starting point of more research on testable generators.

1.5 Plan for the Thesis

We will give a introduction to Leader Election in Ring algorithms in Chapter 2. We will also define the scope of the domain later in the second chapter. Then in chapter 3 we carry out domain analysis. The intermediate model we put forth to bridge from features to states will be introduced in Chapter 4. Then, in chapter 5, we will explain how to build the integrated algorithm and how to implement it in Rational Rose RT. Chapter 6 discusses the benefits and limitations of our work, as well as problems to be addressed in the future.

Chapter 2

The Domain: Leader Election in a Ring

2.1 More Terminology for This Chapter

Before we explain the problem of Leader Election in a Ring, we need to clarify some basic terminology.

1. **Ring:** In our context, a ring is a network topology in which all **nodes** are connected to a single wire in a circle or point-to-point. There are no endpoints. The wire that connects two nodes is called a **link**.
2. **Neighbor:** In the ring topology, every node with connectivity is called the neighbor node of the other two nodes connected to it by a link.

3. **Port:** Each node has two ports through which the link can connect to other nodes.
4. **Unidirectional Ring:** If the nodes in the ring can only send messages in one direction (i.e., clockwise or counterclockwise), then the ring is unidirectional.
5. **Bi-directional Ring:** If the nodes in the ring can send messages to both directions (i.e., clockwise and counterclockwise), then the ring is bi-directional.
6. **Oriented:** In the ring each node has two ports, labelled as *left* and *right*. If this labelling is globally consistent, the ring will be said to be oriented.[40]
7. **Unoriented:** If the labelling of the ports for the nodes is not globally consistent, that is, if there is no common view of "right" and "left", then the ring is called an unoriented ring.
8. **Uniform:** An algorithm is uniform if the number of processors (" n ") is not known to the algorithm. If n is known, the algorithm is called non-uniform.
9. **Anonymous:** A system is anonymous if processors do not have unique identifiers.[44]If we say a ring is anonymous, it means that the nodes are all unlabelled, that is, they cannot be distinguished from each other by name (or identity).
10. **Synchronous:** Communication (in the form of messages) is synchronous if messages are occurring at the same time, or at the same rate, or with a regular or predictable time relationship or sequence.

11. **Asynchronous:** Communication is asynchronous when it is not synchronized (e.g., by a shared signal such as clock or semaphore) and messages proceed independently of each other.
12. **Entity:** In distributed environment, we call any of the processors that participate in a task an Entity. In our context, "Entity" refers to any of the nodes in a ring. Minimally, an entity can pass messages (i.e., act as a relay). If an Entity is a Candidate, then it can also originate messages and make decisions during the election. (These Entities are autonomous and symmetric.)
13. **Candidate:** here "Candidate" refers to the nodes that not only can pass message (since any "Entity" can) but also originate messages and make decisions during the election.
14. **Terminating a message:** If a node does not pass the message it just received from some neighbor, then we say this node terminated this message.
15. **Stage:** When a node sends a message it creates (as opposed to a message it relays) we say the node originates this message. Algorithms in which a node can originate several messages are called multi-stage. Otherwise we have a single stage algorithm. In a multi-stage algorithm, a stage of a node starts each time this node originates a message.
16. **Defeated:** In a multi-stage algorithm, if a node is no longer eligible to originate messages at the next stage, then we say this node is defeated (and the

node becomes a "relay"). A node that is not defeated remains a candidate.

17. **Initiator:** The entities that start sending messages to other entities when an election starts are called Initiators.
18. **ID:** An Initial Distinct Value (to distinguish between nodes) is sometimes called an identifier or an ID or a global name.
19. **Message Ordering:** We assume messages are delivered in the order in which they arrive. This restriction is called Message Ordering.
20. **Symmetric:** We assume messages are delivered in the order in which they arrive. This restriction is called Message Ordering.
21. **Autonomous:** We assume messages are delivered in the order in which they arrive. This restriction is called Message Ordering.

2.2 The ring election problem

2.2.1 Leader Election

Leader election is a fundamental problem in distributed computing. In a distributed environment, most applications often require a single entity to act temporarily as a central controller to coordinate the execution of a particular task by the entities. This controller or coordinator can be any process that organizes actions of other processes. In some cases, the need for a single coordinator arises from the desire to

simplify the design of the solution protocol for a rather complex problem; in other cases, the presence of a single coordinator is required by the nature of the problem itself.

The problem of choosing such a coordinator from a population of autonomous and symmetric entities is known as ***Leader Election***. [41] Formally, the task consists in moving the system from an initial configuration where all entities are in the same state (usually called ***available***), into a final configuration where all entities are in the same state (traditionally called ***follower***), except one that is in a different state (traditionally called ***leader***).

There is no restriction on the number of entities that can start the election, nor on which entity should become leader. Namely, there is no predefined special entity that can break the symmetry initially. We can think of the Election problem as the problem of enforcing a restriction "Unique Initiator" in a system where actually no such a restriction exists: The potential multiple initiators would first start the execution of an Election protocol by originate and send the messages; the sole leader will then be the unique initiator for the subsequent computation. Since election provides a mechanism for breaking the symmetry among the entities in a distributed environment, it is at the basis of most control and coordination processes (e.g., mutual exclusion, synchronization, concurrency control, etc.) employed in distributed

systems, and it is closely related to other basic computations (e.g., minimum finding, spanning-tree construction, traversal).

Intuitively, the ring appears to be a totally symmetric network topology. In order to choose a leader out of its entities, we must break this symmetry. By adopting the restriction that each entity has a unique ID, it is not difficult to break the symmetry by comparing the value of all IDs. We can either choose the smallest ID as the leader, or choose the ID which is the largest, as the leader. The strategy that chooses the smallest ID to be the leader is called Minimum Finding or Elect Minimum, and the strategy that chooses the largest ID to be the leader is called Maximum Finding or Elect Maximum. In the rest of this dissertation, we will refer almost exclusively to the approach of minimum finding (since the discussion would also readily apply to maximum finding). In fact, we will consider both the Elect Minimum and the Elect Minimum Initiator approaches. Here Elect Minimum Initiator means to elect the node with the smallest ID as the leader, and this election is only performed among initiators.

2.2.2 Restrictions

The election problem is deterministically unsolvable if there is no restriction. More precisely, we need all the entities to be at least all connected and totally reliable.

Then we can capture the standard restrictions as :

R : Bidirectional Links, Connectivity, and Total Reliability.

Let us elaborate briefly. As defined in the previous section, an election algorithm should provide a mechanism for breaking the symmetry among the entities in a distributed environment. In other words, the election problem provides a way to make one entity differ from all the other entities in order to take the role of unique coordinator or central controller. But, unfortunately, election is deterministically unsolvable under restriction **R** ([41]’s Theorem 1.0.1). Namely, there is no deterministic protocol that will always correctly terminate within finite time if the only restrictions are those in **R** . So, additional restrictions and assumptions are needed. A very powerful restriction, which is commonly used, is "Initial Distinct Values" (ID) (also called identifiers or global names, we repeat). The presence of such identifiers will be sufficient to elect a leader. Generally, $\text{id}(x)$ denotes the distinct value of x . The use of this additional assumption is so frequent that the set of restrictions **$IR = R \cup \text{ID}$** is called the **Standard Set** for election.

As previously mentioned, all ring election algorithms need some system restrictions to execute and terminate correctly. Let’s briefly first look at some of these algorithms (hereafter called protocols and their restrictions).

1. Protocol "All the Way" (Lelann's algorithm)

Restrictions: $RI^1 \cup Ring$

2. Protocol "Asfar" (Chang and Robert's algorithm)

Restrictions: $RI \cup Ring^2$

3. Protocol "ProbAsfar"

Restrictions: $RI \cup Ring$

4. Protocol "Controlled Distance"

Restrictions: $RI \cup Ring$

5. Protocol "Stages" (Franklin's first algorithm)

Restrictions: $RI \cup Ring$

6. Protocol "Stages* (stages with feedback)" (Santoro's algorithm)

Restrictions: $RI \cup Ring$

¹ $IR = R \cup ID$. R: Bidirectional Links, Connectivity, and Total Reliability

²Protocol "All the way" and "As far" can work in both unidirectional and bidirectional rings with the same cost.

7. Protocol "Alternate"

Restrictions: RI U Ring U Message Ordering

8. Protocol "UniStages"

Restrictions: Connectivity U Total Reliability U Unidirectional U Ring

9. Protocol "UniAlternate"

Restrictions: Connectivity U Total Reliability U Unidirectional U Ring U
Message Ordering

10. Protocol "MinMax" (Peterson's algorithm)

Restrictions: Connectivity U Total Reliability U Ring

2.2.3 Leader Election in Rings

The Ring topology

We will now consider a network topology that plays a very important role in distributed computing: the ring, sometimes called loop network. A ring consists of a single cycle of length n . In a ring, each entity has exactly two neighbors, (whose associated ports are) traditionally called left and right. As we defined earlier, if the

sense of "left" and "right" are globally consistent, then we have an oriented ring, otherwise it is an unoriented ring. We will denote the ring by $R = (x_0; x_1; ; : : : ; x_{n-1})$. Next, we consider the problem of electing a leader in the ring network topology under the Standard Set of restrictions for election: IR= Bi-directional links, Connectivity, Total Reliability, Initial Distinct Values.

2.3 The selection of algorithms

In order to scope the domain, an extensive literature review was performed on leader election in a ring. In this section, we briefly describe the boundaries of the domain at hand, then introduce several algorithms of the domain that will be used as examples in the other chapters of this thesis.

In 1977, G. Lelann[34] published a first ring election algorithm in his paper "Distributed systems: toward a formal approach". Since then, leader election algorithms for rings have received considerable attention in distributed computing research. Ernest J. H. Chang and R. Robert introduced the "Chang and Robert's" algorithm[12] in their paper in 1979, which improved message cost and time cost with respect to the original LeLann's algorithm. In 1980, D.S. Hirschberg and J.B. Sinclair described a new algorithm that reduced the worse case message cost to $6.31n \log n + O(n)$ [24]. Later, in 1982 three similar algorithms were published:

”stages” [19], ”stages with feedback” [17] and ”stages in unidirectional ring” [38]. (We will introduce protocol Stages later.) Since 1982 several other algorithms were developed for the unidirectional ring [23],[28],[38].

Our goal in this thesis is to investigate and model a selection of existing algorithms in this domain, generate an executable integrated model for the entire domain, and use this executable model to generate some members of the domain (be they existent or new algorithms). The first step is one of conceptualization: it requires a deep understanding of the domain as a whole. On this issue, little can be said: the learning curve and the ability to integrate is idiosyncratic as previously mentioned. Though we will present an integrated model that addresses all the algorithms we discuss, we will illustrate the discussion by focusing mainly on three specific algorithms. They are: Protocol ”All the Way” (LeLann’s algorithm); Protocol ”Asfar” (Chang and Robert’s algorithm); and Protocol ”Stages*(stages with feedback)” (Peterson’s improved algorithm). In the rest of this section, we will introduce those three election algorithms briefly.

Protocol ”All the way” (LeLann):

Under the restriction of $RI \cup Ring$, each entity originates a message containing its id. Each message will travel along the ring (forwarded by all the other entities).

Entities only forward messages carrying values other than their own IDs: once the message with $id(x)$ arrives at x , the message is no longer forwarded. Hence, each entity will eventually see everybody else's IDs, including the minimum value (id). Each entity will thus be able to determine whether or not it has the (unique) minimum id and also know who is the leader. As usual, the number of nodes in the ring is unknown. Instead, this algorithm uses a counter at each entity to keep track of how many different IDs are received, and a counter in each message to keep track of how many entities it has passed. Consequently, each entity can determine the total number of nodes in the ring, as well as figuring out the leader.

Protocol "AsFar":

Under the restriction of $RI \cup Ring$, each entity originates a message containing its id . Each message will travel along the ring (being forwarded by all the other entities). Entities only forward messages carrying a value smaller than their own IDs. Once the message with $id(x)$ arrives at $id(y)$ if $id(x) \geq id(y)$, the message is no longer forwarded. Thus only the message with the minimum value (id) will arrive back at the entity that has the same value as it contains. Any entity that receives back the message that it originated becomes the leader. Then the leader sends a notify message to one of its neighbors. The notify message is forwarded along the ring until it comes back to the leader.

Protocol Stages with Feedback:

A candidate x sends a message in both directions carrying its identity; these messages will travel until they encounter another candidate node. By symmetry, entity x will receive two messages, one from the "left" and one from the "right"(independently of any sense of direction). Each time one node received a message, it will compare the ID of itself and the one from the message. Here we bring the idea of **Feedback**. The node will send a feedback message to the message's originator. The feedback consist of "positive feedback" and "negative feedback". If the ID in the message is smaller than the node's ID, then a positive feedback will be sent, otherwise a negative feedback will be sent. If and only if one node received two positive feedback, it can survive this round, and originate again in the a new stage. Otherwise it will then become defeated if at least one of them is negative feedback (that means the ID of that identity is smaller than its own). Finally, if the received IDs are from its own, it becomes leader and notifies all entities to terminate.

In this algorithm, a defeated node will forward any received election message, and each non-initiator will automatically become defeated upon receiving an election message. See example in Figure 2.1, candidates with IDs 2, 5 and 8 will not

Figure 2.1: The Example for Protocol "Stages with Feedback"

send any feedback; of these three, only candidate with id=2 will enter next stage. The fate of entity with id 7 depends on its other neighbouring candidate, which is not shown; so, we do not know whether it will survive or not. If a node sends a "Feedback" message, it knows that it will not survive this stage. This is the case, in the example, of the entities with IDs 6, 9, 10, 11. Some entities, however, do not send any "Feedback" and wait for a "Feedback" that will never arrive; this is for example the cases of the entities with IDs 5 and 8. How will such an entity discover that no "feedback" is forthcoming, and it must become defeated. The answer is fortunately simple. Every entity that survives stage i (e.g., the node with $id=2$) will start the next stage; its Stage message will act as a negative feedback for those entities receiving the message while still waiting in stage i . More specifically, if while waiting for a "feedback" message in stage i , an entity receives an electing message (clearly with a smaller id) in stage $i+1$, it becomes defeated and forwards the message.

We will introduce the feature model for leader election in ring domain in Chapter 3. For now we just list the "Feature algorithms relationship diagram" here as the summarize for the introduction of our argument domain in this chapter. We will use and explain this table Table 2.1. later in Chapter 5.

2.4 Initial Conceptual Considerations

In the ring leader election domain, the nodes communicate with each other only by sending messages to each other [20]. It does not matter which mechanism each algorithm uses, the algorithm must make one node in the environment change into a different state from the others. There is no predefined node to break the symmetry in the topology. That is, given a specific topology, the relationships between the nodes of that topology must remain stable and predictable. For example, in a ring, each node has two and only two neighbors. In the same vein, we remark that all algorithms in the domain:

1. have the same requirements;
2. run in the same environment;

	Protocol "All the way" (Lelann's)	Protocol "AsFar" (Chang and Robert's)	Protocol Control	Protocol Stage	Protocol Stage with Feedback	Protocol Alternate	Protocol Unidirectional Stages	Protocol MinMax	Protocol Unidirectional Alternate
F1V1									
F1V2									
F2V1									
F2V2									
F3V1									
F3V2									
F4V1									
F4V2									
F4V3									
F5V1									
F5V2									
F6V1									
F6V2									
F6V3									
F7V1									
F7V2									
F7V3									
F7V4									
F8V1									
F8V2									
F9V1									
F9V2									
F10V1									
F10V2									
F11V1									
F11V2									
F12V1									
F12V2									

- * means either of the feature values is chosen will make the corresponding protocol has the same behavior.
- * means this feature value is chosen by the corresponding protocol in the table.

Table 2.1: Feature Algorithms relationship.

3. use the same way to communicate between nodes.

The only difference is that they use different mechanism to break symmetry in order to choose the leader. From this viewpoint, clearly, the conclusion is that algorithms for Leader Election in a Ring can form a family (i.e., a domain), which we will call the **Leader Election in Ring** domain.

Chapter 3

An Integrated Leader Election

Algorithm

3.1 Functional requirements and Non-functional Requirements

3.1.1 Functional requirements

FR1: Each candidate x_i in the ring can originate a message that contains at least this entity's ID: $id(x_i)$.

FR2: All the entities must pass messages, regardless of originator, when appropriate. That is,, any message can travel along the ring before it is terminated.

FR3: The messages can be terminated by a certain kind of strategy.

FR4: It must be possible to elect a leader (who ends up with a state distinct from the one of all other nodes).

FR5: All the other nodes in the ring will be notified by the new leader, if needed. System assumption R: Bidirectional Links, Connectivity, and Total Reliability. ID: Initial Distinct Values Standard set for election: IR= R?ID=Bidirectional links, Connectivity, Total Reliability, Initial Distinct Values

3.1.2 Non-functional requirements

There are two non functional requirement in the domain:

NFR1: Minimize total number of messages to be used.

The system wants to have as few messages as possible during the election.

NFR2: Minimize time cost

The system wants to elect the leader as quickly as possible.

3.2 Initial feature modeling

3.2.1 Choosing Features

Before we explain the feature model we propose for the target domain, we need to address how to go about finding features.

Strategies for identifying features include both top-down and bottom-up strategies.[32]

The most frequent suggested method is to look for important domain terminology that implies variability. In our case, we used a bottom-up strategy to build our feature model. That is, we first looked at everything that was different from one algorithm to the next one and from this analysis we extracted features (i.e., variation points). In other words, features values were grouped into features. We then organized the features into a hierarchy. We decided to break the domain into three groups of features, as will be explained shortly. It is important to repeat that we do not claim the proposed feature model is complete. But, as we will demonstrate, it was sufficient to build an executable integrated model, a fundamental goal of this dissertation. Let us now turn to a discussion of the proposed features.

3.2.2 Distinguishing Features and Parameters

It is important to distinguish between a parameter and a feature. A parameter defines a meaningful attribute input to the system. Contrary to a feature, we view a parameter not as a functional unit but as a datum (e.g., length or weight). A feature may have parameters associated with it. These parameters may fully or only partially define a feature. For example, for the feature *Send Message Strategy* (introduced below), we use two possible feature values: *Randomly choose direction* and *Send message to both neighbors*. But only one parameter is related to feature *Send Message Strategy*, namely the parameter *sendDirection*, which can be set to "right" or "left" or "both". Thus, this parameter does affect the behavior of that feature (without fully defining it).

3.2.3 Selection of Features and Parameters

List of System Parameters

- Minimum Value: Record the smallest ID that one node received. It is used when minimum electing strategy is used.
- Maximum Value: Record the biggest ID that one node received. It is used when maximum electing strategy is used.
- Counter: This is a parameter used for messages to count how many nodes has the message passed. (contains in the message, controlled by nodes)

-Number of Messages Passed by: Used for each node to count how many messages have passed through it.

-Id: This stands for the unique identification of each node. -Stage Number: Used for each node to record which state it is in.

-Distance Limitation: Used to control how long can each message go (contains in the messages). -Message type: Used for node to distinguish which type of message it received. It could be electing message, feedback message, notify message (contains in the messages).

-Number of nodes in the ring: Used for each node to record how many nodes in the network. This parameter will only be used when the algorithms do not use send notify message strategy to do the election.

List of Features

-Ring type; -Direction of The Ring; -Election Scope;

-Election Strategy; -Stages; -Send Message Strategy;

-Terminate Passing Message Strategy; -Keep Message Order Strategy;

-Stage Advance strategy; -Send Message Mode; -Notification;

-Node Behavior;

In order to simplify our explanation and figures, we will give the Feature and symbol corresponding table in Table 3.1. Each of these features is explained in the

next section. In the following diagram, we capture which parameters affect which features.

3.3 Details of Features

3.3.1 Features and Feature Values Definition

After investigating each of the selected algorithms, we grouped all features into three groups: Topology, Node Behavior and Protocol 3.1. Under each feature group, we have several features and feature values. As the reader scrutinizes the following descriptions, it will be important to remark that algorithms do not map on a single feature but a set of features. In other words, each feature is a logical functional unit potentially reused by several different algorithms. And each algorithm involves several features. Mapping features onto algorithms, as well as addressing feature interactions will follow our descriptions of the different features and of their possible values.

Topology Group

The Topology group consists of two features: Ring Type and Direction of Ring. Ring Type is a feature to describe whether each node in the ring can send messages

Feature 1	Ring type	F 1
Feature value 1-1	Unidirectional Ring	F 1 V 1
Feature value 1-2	Bidirectional Ring	F 1 V 2
Feature 2	Direction of The Ring	F 2
Feature value 2-1	Oriented Ring	F 2 V 1
Feature value 2-2	Unoriented Ring	F 2 V 2
Feature 3	Election Scope	F 3
Feature Value 3-1	Among all nodes	F 3 V 1
Feature Value 3-2	Among initiators	F 3 V 2
Feature 4	Election Strategy	F 4
Feature Value 4-1	Elect minimum	F 4 V 1
Feature Value 4-2	Elect maximum	F 4 V 2
Feature Value 4-3	Elect "min-max"	F 4 V 3
Feature 5	Stages	F 5
Feature Value 5-1	Single stage	F 5 V 1
Feature Value 5-2	Multiple stages	F 5 V 2
Feature 6	Send Message Strategy	F 6
Feature Value 6-1	Randomly choose direction	F 6 V 1
Feature Value 6-2	Send message to both neighbors	F 6 V 2
Feature Value 6-3	Send to the open direction	F 6 V 3
Feature 7	Terminate Passing Message Strategy	F 7
Feature Value 7-1	Complete	F 7 V 1
Feature Value 7-2	Block	F 7 V 2
Feature Value 7-3	Originator controls the distance	F 7 V 3
Feature Value 7-4	To the next candidate	F 7 V 4
Feature 8	Keep Message Order Strategy	F 8
Feature Value 8-1	Queue management	F 8 V 1
Feature Value 8-2	"Ignore" management	F 8 V 2
Feature 9	Stage Advance strategy	F 9
Feature Value 9-1	Do not use feedback	F 9 V 1
Feature Value 9-2	Using feedback mechanism	F 9 V 2
Feature 10	Send Message Mode	F 10
Feature Value 10-1	Simultaneous send	F 10 V 1
Feature Value 10-2	Alternate send	F 10 V 2
Feature 11	Notification	F 11
Feature value 11-1	Need to Notify	F 11 V 1
Feature value 11-2	Do not Need to Notify	F 11 V 2
Feature 12	Node Behavior	F 12
Feature value 12-1	Same Behavior	F 12 V 1
Feature value 12-2	Different Behaviors	F 12 V 2

Table 3.1: Identifiers for feature values

Figure 3.1: Top Level Feature Diagram for Leader Election in Ring

into two directions or only one. It has two possible feature values. If a ring is unidirectional, then each node has one neighbor from which it receives messages, and another to which it sends messages. If a ring is bidirectional, then nodes can send messages in both directions. This feature directly relates to the total number of messages exchanged during an election.

Direction of Ring addresses whether or not a ring has a consistent sense of direction across its nodes. If the ring is oriented, then, we repeat, sending a message to your "right" means the same for all nodes. In this case, all messages will travel in only one direction: clockwise or counter-clockwise. Conversely, the ring is called un-oriented if the nodes in the ring have their own definition of "right" or "left". In this case, all messages can travel in different directions. The fact that the ring is not oriented means that when two entities send a message to their "right" neighbors, they might send it in different directions.

The following figures:

Feature 1 Ring type

Feature value 1-1 Unidirectional Ring

Feature value 1-2 Bidirectional Ring

Feature 2 Direction of The Ring

Feature value 2-1 Oriented Ring

see Figure 3.2

Figure 3.2: Oriented Ring

Feature value 2-2 Unoriented Ring

see Figure 3.3

Figure 3.3: Unoriented Ring

A feature model for the Topology features is given in figure 3.4.

Protocol Group

The Protocol group has a set of or-features with five mandatory features and three optional feature in it. See Figure 3.5. The following is the brief description of all features and feature values:

1. Feature 3 Election Scope

- Feature Value 3-1 Among all nodes: the election is to consider all the nodes

Figure 3.4: Feature Diagram for Leader Election in Ring-Topology Group

- Feature Value 3-2 Among initiators: Sometimes the election does not need to be executed among all the entities in the ring. Instead, only some of the entities will be involved in the election. Such participants will send initial messages. In this case, the nodes that do not send initial message are called relay nodes.

2. Feature 4 Election Strategy

- Feature Value 4-1 Elect minimum: the node with the smallest value is elected as the leader
- Feature Value 4-2 Elect maximum: the node with the largest value is elected as the leader
- Feature Value 4-3 Elect "min-max": the algorithm will execute a "minimum" or "maximum" election in its different stages.

3. Feature 5 Stages

- Feature Value 5-1 Single stage: A node makes only one attempt to become leader by sending a message containing its id only once.
- Feature Value 5-2 Multiple stages: A node makes several successive attempts to become a leader (until it wins or is defeated). Each time a node originates a message, it starts a new stage. The nodes of the ring use several stages to finalize the election.

4. Feature 6 Send Message Strategy

- Feature Value 6-1 Randomly choose direction: if a node can send messages to both of its neighbors, then each time it must send a message, it randomly chooses one neighbor as the destination.
- Feature Value 6-2 Send message to both neighbors: each node sends each message to both of its neighbors.
- Feature Value 6-3 Send to the open direction: each node sends each message to the unique open direction.

5. Feature 7 Terminate Passing Message Strategy

- Feature Value 7-1 Complete: each message travels along the ring until it

comes back to its originator

- Feature Value 7-2 Block: each message travels until it arrives at a node with a smaller ID in "Elect minimum" strategy, or until it arrives a node with a larger ID in "Elect maximum" strategy
- Feature Value 7-3 Originator controls the distance: the originator of the message determines how many nodes the message should traverse along the ring
- Feature Value 7-4 To the next candidate: the message will travel along the ring until it arrives at the next candidate (that is, it skips over relay nodes).

6. Feature 8 Keep Message Order Strategy

- Feature Value 8-1 Queue management: use a queue to maintain the message arrival order, when different links of the ring have different speeds.
- Feature Value 8-2 "Ignore" management: only process the message with the largest stage number, and ignore all the messages from the lower stage, when different parts of the ring have different speed.

7. Feature 9 Stage Advance strategy

- Feature Value 9-1 Do not use feedback: Do not need or use feedback message to decide whether a candidate should advance to the next stage.
- Feature Value 9-2 Using feedback mechanism: If a message successfully

reaches its target (next candidate or reach the distance its originator controlled), then this target will send a feedback message back the message originator; the originator will then decide whether to advance to the next stage accordingly.

8. Feature 10 Send Message Mode

- Feature Value 10-1 Simultaneous send: each node sends the message to both sides along the ring at the same time
- Feature Value 10-2 Alternate send: each node sends the message to one side in one stage, then the other side in the next stage.

Node Behavior group

The Node Behavior group has two mandatory features: Notification and Behavior. After the leader is elected, all the nodes of the ring should stop electing immediately. There are two ways to make the rest of the ring get into some terminate stage. One consists in having the leader send a notify message to make all the other nodes know who has been elected as leader. We will call this "Need to Notify". The other way is to have each node decide by itself a) whether or not it is the leader, and b) if it is not the leader, figure out who the leader is. We call this "Do not need notify".

Figure 3.5: Feature Diagram for Leader Election in Ring-Protocol Group

Summarizing:

Feature 11 Notification

Feature value 11-1 Need to Notify

As explained above. Relevant to protocols AsFar, Control, and Stages.

Feature value 11-2 Do not need to Notify

Again, as explained above. For example: Protocol All the Way.

Feature 12 Node Behavior

In the election problem, some nodes will change their behavior during execution. For example, when using a minimum election strategy, assume a node X receives a message with a smaller id than itself. Then node X is defeated and changes its behavior accordingly: it becomes a relay node.

Feature value 12-1 Same Behavior

During the entire execution of a leader election, each node keeps the same behavior until it terminates.

For example: Protocols All the Way and AsFar.

Feature value 12-2 Different Behaviors

During the execution of leader election, each node will possibly change its behavior: behavior will be modified based on which messages it received. Usually, a node can be candidate, or be defeated and thus act as a relay node. (see Feature 3)

For example: Protocol Control, Stages, and Stages *.

See Figure 3.6.

Figure 3.6: Feature Diagram for Leader Election in Ring-Node Behavior Group

The feature diagram for the entire domain is shown in Figure 3.7

3.3.2 Feature Interactions - Combination Rules

F1 → F2: Restriction

If F1V1 then only F2V1

If F1V2 then F2V1 or F2V2

-Unidirectional Ring (F1V1) must be Oriented (F2V1), and Bi-directional (F1V2)

Ring could be Oriented (F2V1) or Un-oriented (F2V2).

F1, F4 → F6 Joint (Combined) Precondition

Figure 3.7: Feature Diagram for Leader Election in Ring Topology Domain

If F1V1 and F4V2 then must F6V1

- When Multiple Stages (F4V2) is used in a Unidirectional Ring (F1V1), then Random Choose one Direction (F6V1) must be chosen.

F4 → F7, F8, F9

F4 → F7 Precondition

F4 → F8 Restriction

F4 → F9 Restriction

Only if F4V2 then must have F8 and F9 at the same time, maybe F7

-Only when Multiple Stages (F4V2) is used, both Notify (F8) and Send Message Mode (F9) must be taken into consideration at the same time. And which way to Keep Message Order Strategy (F7) should be thought of, only if the message in different stages possibly arrive without order (i.e., message in higher stage arrived before the lower stage because of the different speed of every part of the ring.).

F1, F4 → F3 Joint

(Combined) Precondition Only if F1V1 and F4V2 then F3V5 is possible

-Only when Multiple Stages (F4V2) is used in a Unidirectional Ring (F1V1), can Elect "min-max" (F3V5) of the Election Strategy (F3) possibly be used.

F1 → F5 Precondition

Only if F1V2 then F5V2 is possible

-Only when the election is executed in a Bi-directional Ring (F1V2), can a message be sent to both of its neighbors. More precisely, only here can Send Message to Both Sides of Originator's Neighbors (F5V2) be used.

F5 → F9 Restriction

If F5V2 then must F9 -When the election is executed in a Bi-directional Ring (F1V2), and the messages are send to both sides of the originators' neighbors (namely, Send Message to Both Sides of Originator's Neighbors (F5V2) strategy is chosen), then Send Message Mode (F9) must the taken into the consideration. Either Simultaneous Send (F9V1) or Alternate Send (F9V2) may be selected.

F5 → F12 Restriction

If F5V2 then must F12V2 -When the algorithm uses multiple stage to elect leader, then the nodes must behave differently according to the result of each stage or the sub-step in each stage.

Chapter 4

Beyond Feature Model

4.1 State Machine Model

4.1.1 Why State Machine?

Feature modeling helps us scope and make explicit the conceptual framework of the domain. Not only can we use features to describe the domain but also we can consider combinations of feature values to express family members, that is, concrete algorithms in the domain. But a feature model and its associated combination rules do not provide a solution to another one of our goals, namely, executability. Generative programming typically leaves a large semantic gap between feature models and the generator that can produce executable family members. The gap would be even larger if the generator had to reuse the semantics of some simulator for distributed algorithms. And the idiosyncrasies of such simulators may make the integration of

a family of algorithms into a single executable model quite difficult (since some algorithms could be using mechanisms simply not available in the simulator at hand). Our solution to the requirement of executability rests on the key observation that leader election algorithms are generally state-based. Thus, it is quite natural to ask whether it is feasible to integrate the family of election algorithms we have selected into a single state machine. Doing so would be a first we believe. And it would offer two immediate advantages: 1) there are commercial CASE tools capable of simulating state machines (e.g., ROSE-Real Time from IBM) and 2) state machines are highly testable models[11]. In fact, ROSE-RT already offers some testing services. It is not our intention here to discuss testing but rather to summarize the concepts and advantages of state machines in order to then study in detail the integrated one we developed.

4.1.2 What is State Machine?

Binder gives the following definition of a state machine in his seminal book[11] on object-oriented testing: "*A State Machine is a system whose output is determined by both current and past input. The effect of previous inputs is represented by a state.*". A state machine is an engineering application of mathematical models known as finite automata. Finite automata were first applied to the design of switching circuits in the 1950s and continue to serve as essential tools for the design and testing of

digital logic. State machines were applied to software design problems at beginning of the 1960s and became routinely used by the end of the 1970s, in software development methods regrouped under the umbrella term "structured analysis and structured design". With the advent of object-oriented software engineering, instead of developing a state machine for a whole system, researchers advocate using state machines to capture the detailed behavior of each class that makes up the system. The importance of state machines can be confirmed in current development practices by remarking that it is a pivotal aspect of the Unified Modeling Language(UML) [11], the dominant notation for software modeling.

A state machine is an abstraction composed of events(inputs), states and actions (outputs). It can be conceptualized as a directed graph. There is a finite number of states, and each state has transitions to zero or more states (including itself). There is an input string to the state that determines which transition is followed.[39, 9, 8, 25, 3] A system is said to exhibit **state-based behavior** when identical inputs are not always accepted and, when accepted, may produce different outputs[11].

A state machine is a model of computation consisting of a set of states, a start state, an input alphabet, and a transition function that maps input symbols and current states to a next state. Computation begins in the start (or initial) state with an

input string. It changes to new states depending on the transition function. There are many variants. For instance, machines may have actions (outputs) associated with transitions (Mealy machine) or with states (Moore machine). Also, one may have multiple start states, transitions conditioned on no input symbol (i.e., a null transition), more than one transition for a given symbol and state (i.e., a non-deterministic finite state machine), one or more states designated as accepting states (recognizer), etc.[4] A state machine has four building blocks:

- **State.** An abstraction that "summarizes the information concerning past inputs that is needed to determine the behavior of the system on subsequent inputs" [26].
- **Transition.** An allowable two-state sequence. A transition is caused by an event. It may result in an output action and must specify an accepting state and a resultant state.
- **Event.** An input or and interval of time.
- **Action.** The result or output that follows an event.

The system is taken from one state to another by a **transition**. The **initial state** is the state in which the first event is accepted. The model specifies the initial state – it does not need to be entered by a transition. Also, a **final state** is one in which the machine stops accepting events.

State Transition Diagrams

A state transition diagram is a graphic representation of a state machine. Nodes

represent states. Arrows (directed edges) represent transitions. The annotations on the edges represent events and actions. An example is given in 4.1. Here nodes

Figure 4.1: State Transition Diagram example

Valve Shut, **Valve Open** represent the two states of the model of an automatic re-filling mechanism for a tank of liquid. **Tank Full**, **Float Low** are two events that trigger the transitions. And **Close Valve** and **Open Valve** are two actions occurring on the transitions.

4.2 An Intermediate Model to Fill the Gap between Feature Model and State Machine Model

4.2.1 Motivation

The previous discussion makes it clear that there is almost no relationship between a Feature Model and a State Machine Model. The question then is how to go from the former to the latter. We suggest an intermediate model we will call a ***Feature-State-Decision-Flow Chart (FSDF)***, which proceeds from the mixing together the notions of a **Decision Tree** and a **Flow Chart**. Let us briefly discuss each of these.

4.2.2 Decision Trees and Flow Charts

A **Decision Tree** is a representation of a decision procedure for determining a class label to associate with a given example. At each internal node of the tree, there is a test (question), and a branch corresponding to each of the possible outcomes of the test. At each leaf node, there is a class label (answer). Traversing a path from the root to a leaf is much like playing a game of twenty questions[6].

Decision trees have a great many uses, particularly for solving problems that can be cast in terms of producing a single answer in the form of a class name. For example, one can build a decision tree that could be used to answer a question such

as 'Does this patient have hepatitis?' The answer may be as simple as 'yes' or 'no'. Based on answers to the questions at the decision nodes, one can find the appropriate leaf and the answer it contains.

Decision trees are constructed from examples that are already labelled. For example, if one has established for a variety of patients with varying attributes, which of them do and do not have hepatitis, then these examples can guide the tree construction process. There is much ongoing research on decision tree induction. The subject is studied within several disciplines and is accessible from publications such as the Journal of Machine Learning journal and proceedings of Machine Learning conferences. Here is an example of a decision tree 4.2[2].

Figure 4.2: Decision Tree example

A **Flow Chart** is a formalized graphical representation of some program logic sequence, work or manufacturing process, organization chart, or similar formalized structures. In computer programming, Flow Charts were used 30 years ago to describe each processing path in a program (i.e., the main program and various sub-routines that could be branched to). Programmers were admonished to always Flow Chart their logic rather than carry it in their heads or bury it in code. With the advent of object-oriented programming and its emphasis on visual development tools, the traditional program Flow Chart has been more or less abandoned. However, there are newer forms of Flow Charts that can be used for data and class modeling in object-oriented programming.[5] Here is an example of Flow Chart 4.3.[1]

4.3 Feature-State-Decision-Flow Chart

Our proposed **Feature-State-Decision-Flow Chart(FSDF)** is a graphical representation we use to describe the relationship between features, their values and the states of the system. It is a model that aims at integrating Feature Models, states in State Machines, Decision Trees and Flow Charts. The FSDF Chart is a tree like graph that consists of a root, nodes, leaves and arrows. The **root** of the tree is the initial state in the related state machine. It is represented by a full circle. Each **node** is either a **decision point** or a **state** (but not the initial one nor a final state). A node is represented by an empty circle. Each **leaf** is a final state in the related

Figure 4.3: Flow Chart Example

state machine, and is represented by a double empty circle. Each **arrow** points to either a leaf or a node. The arrow is either a **decision branch** or a transition. If the arrow comes from a **decision point**, then it is a decision branch. If the arrow comes from a state, then it is a **transition**. An event and an action must be associated with a transition. Each decision point represents a choice among feature values. More precisely, the decision point spells out which configuration of feature values leads

Figure 4.4: FSDF Chart Notation

to which other decision point or to which new state. A decision point can be as simple as a decision which has a "Yes" or a "No" on its the decision branch. We give an example below 4.4. Our task is now clear for the next chapter. We will first give the Feature-State-Decision-Flow Chart we created for the feature model presented in chapter 4. We will then explain how this diagram organizes our executable state machine and finally, we will discuss at length the states, transitions, and code of the latter.

Chapter 5

An Integrated Executable Model

5.1 Towards An Integrated Executable Model

As stated in the first chapter, software engineering can be viewed as attempting to bridge the gap between software requirements [35],[33] and the resources that can be reused to create a solution that can be implemented to satisfy these requirements. In system family engineering, this suggests the separation of the problem space from the solution space[18]. In the previous chapters, we briefly discussed the Functional and Non-functional requirements for leader election algorithms in rings, developed a feature model for that domain and started bridging this domain (i.e., the problem space) into a specific solution space, namely a state machine. To do so, we suggested the use of a *FSDF Chart*. But we still do not have an executable model. Such a model would ideally input some parameters and feature values from

its user and produce the corresponding family member in the domain. This is usually called a generator in system family engineering. It is typically defined as *"a program that takes a higher-level specification of a piece of software and produces its implementation"* [13].

Czarnecki and Eisenecker [32] present a generator as the bridge from Domain Analysis and Design (in the problem space) to Domain Implementation and Application Engineering (in the solution space). But it is commonly accepted that such a generator is not as much a bridge as an artifact in the solution space. And there is a large semantic gap between the two spaces.

DeBruin and vanVliet have suggested a Feature-Solution (FS) graph to explicitly tie features that address functional and non-functional requirements to scenarios in the solution space [14]. Their approach has the advantage that scenarios can be used to produce system-level tests. But they are not interested in producing a generator.

As previously mentioned, ultimately we, as a research group, are interested in the issue of quality engineering across domains, and in particular in the generation of tests across the domain. It is specifically from this angle that we choose to use state machines for our solution space. As Binder emphasizes at length in chapter 7 of his seminal book [11], a state machine is what he calls a **test-ready model**, that

is, one that contains enough information to produce test cases automatically. While the issue of test case generation out of a state machine lies beyond the scope of our dissertation, we do reiterate the reasons for our adoption of such state machines for our solution space:

- 1. State machines are a testable model.**
- 2. State machines do fit the state-based nature of Leader Election algorithms.**
- 3. We readily have access to an industrial-strength case tool (IBM's Rational Rose Real Time- hereafter ROSE-RT) that supports both simulation and some test case generation.**

The adoption of ROSE-RT does impose a restriction however on our work. As it stands, this tool does not support the sort of C++ meta-programming [32] used for the creation of generators proper. Put another way, it does not have the ability to capture a state machine whose compilation (not execution) would generate a state machine (which is really what a generator should do). Instead, the model we propose for the solution space is a state machine which acts as a behavior generator through its execution (as opposed to its compilation). That is, a family member of the domain (i.e., a specific algorithm) is not generated per se, but instantiated at run-time through the execution of the integrated state machine we propose in this chapter. In other words, it is the execution of our integrated state machine that produces the specific algorithm the user will have chosen through her selection of parameters and features. The values of those parameters and features are used on

the transitions of the proposed state model to encode the different behavioral variants (i.e., features) this model supports. The beauty of the approach is that the state machine model captures the behavior of all nodes. Then each node at run-time has its behavior proceed from the messages it receives.

The FSDF Chart proposed in the previous chapter gets us started about what could be the states of the integrated state machine we want. But to develop this executable model per se we must first study the semantics of ROSE-RT.

5.2 A Crash Course in ROSE-RT

Rational Rose Real Time is a graphical tool for programming real time systems. It is a comprehensive visual development environment that enables developers to design applications at a high level.

It is a software development environment tailored to the demands of real-time software. Developers use Rose Real Time to create models of a software system based on the Unified Modeling Language (UML) constructs, then to generate the implementation code, and compile, then run and debug the application.

Using Rose Real Time, developers work at a high level of abstraction speci-

ifying behavior in state diagrams and communication relationships in collaboration diagrams. This is a natural and logical evolution from computer languages. Just as third generation language provided greater productivity than assembly language coding, visual development tools provide significant productivity gains over current third generation languages [16].

In addition to supporting the core UML constructs, Rose Real Time uses the extensibility features of the UML to define some new constructs that are specialized for real-time system development. These new constructs allow code generation of elements that can use the services provided in the Services Library, such as concurrent state machines, concurrency, message passing, and timing services. Many industrial real-time projects must implement most of the above services.

Let us take a quick look at some basic definition in the tool:

Capsules

- A capsule is a stereotype of a class.
- A capsule provides built-in support for light weight concurrent objects.
- A capsule is a highly encapsulated object using message based communication to other capsules via its port objects.
- Capsules can aggregate other capsules.

Protocols

A protocol defines the set of messages exchanged between a set of capsules. Messages are defined from the perspective of both the receiver and the sender. There are therefore 2 different perspectives for a protocol, which we call protocol roles. Protocol roles represent the communication from the perspective of one participant in the communication scenario. Messages that are sent between capsules contain a required signal name (which identifies the message), an optional priority (relative importance of this message compared to other unprocessed messages on the same thread), and optional application data.

Ports

Ports are objects whose purpose is to send and receive messages to and from capsule instances. They are owned by the capsule instance in the sense that they are created along with their capsule and destroyed when the capsule is destroyed. To specify which messages can be sent to and from a port, a port realizes a protocol role. The protocol role is the specification of a set of the messages that can be received (in) and sent (out) from the port. The protocol role essentially defines the port type.

State Diagrams

State diagrams use the same notation as defined in the UML. They can be used

to generate source code. They make up the behavior of capsules. All trigger events for transitions are defined by a port and signal pair. A capsule's behavior is therefore based on the receipt of messages. Final states are not allowed in capsule state diagrams. And junction points do not support the continuation kind attribute; that is, if a transition is not continued, it defaults to history (except for internal transitions). Semantically, the state machines of ROSE-RT are State Charts [21],[15].

Capsule Structure Diagrams

A new diagram has been introduced in the UML to specify the capsule's interface (ports) and its internal composition (capsule roles). The diagram is called a capsule structure diagram and it allows the user to specify the communication relationships between capsules. See Figure 5.1

Though a multitude of details could be added here with respect to the extensive semantics of ROSE-RT, we instead refer the reader to the excellent tutorials offered in that tool, as well as to its Help facility which plays the role of a user's guide.

5.3 Two Individual State Charts for Two Algorithms.

Recall the protocols "Asfar" (Chang and Robert's algorithm), "All the way" (LeLann's algorithm) we introduced in chapter 2. We are going to introduce the State Charts

Figure 5.1: Class Diagram with Capsules

of those two algorithms individually in this section, and explain how to integrate the two state machines of these two algorithms in the next section. We will consider the election of a minimum.

The nodes participating in the leader election by running protocol "Asfar" will all start from the initial state "Asleep". Then the system timeout wakes the nodes up. We should notice here that the network can be synchronous or asynchronous. This algorithm works in both cases. For now, we do not care how the nodes cooperate with each other. That is, as long as each node can execute each action within a finite amount of time, we will not require that the nodes have to do everything at the same time. Now we will express the algorithm from the viewpoint of a single node in the ring network, since all the nodes in the system will behave according to one set of rules. After the node wakes up, it sends an electing message to one of its neighbors. If it receives an electing message that is originated by some other node and the value contained in that message is smaller than its own, then it always passes this message to the next neighbor along the ring. If it receives an electing message that is originated by some other node and the value contained in the message is larger than its own, then it stops passing the message and deletes it. If it receives the electing message originated by itself, then it becomes a leader and sends a notify message to its neighbor. If one node receives a notify message, then it becomes a follower and passes the notify message to the next neighbor along the

Figure 5.2: The State Chart for Protocol "Asfar" (Chang and Robert's algorithm)

ring. There is a special case: a node could be a relay node, which means it will never originate message. In this case, it still does work like nodes, except for the fact it does not originate any message. Once a leader or a follower, a node sends the appropriate message and then stops processing messages. Followers remain in that state, whereas the leader goes to state "terminated". Let's now consider the State Chart in Figure 5.2.

In the figure 5.2, rectangles represents the states, arrows represents the transitions and the text explains the events and the actions happening on the transitions.

The events appear before the slash of a transition, and actions are the part after the slash in the text. The states are: **Asleep, Candidate, Relay, Leader, Follower and Terminate**. Here state Asleep is an initial state; Terminate and Follower are the two final states.

On the transition from state Asleep to Candidate, the event is "Initiator wakes up (time out)". Actually we can control how to wake up one node as we want. The usual way to trigger the node to start being active (wake up) is to set a timer: when this timer expires in a node, it triggers the state Candidate (or waken up). Then the node originates an electing message, which is the action on the transition from state Asleep to state Candidate. There are two transitions from state Candidate to itself. Each transition has its own "event/action" on it. The first transition is triggered by event "Received an electing message from the other node that has a value smaller than its own". The action "Pass this message" is required in this case. The other transition is triggered by another event "Received an electing message from the other node that has a value larger than itself". It is important to notice that the action on this transition is nothing! Indeed, according to the algorithm, when a node receives a message that contains a larger ID, then it will "block" the message from going any farther (i.e., the node will not keep passing the message). So the action on this transition is "No action". From this example, we can come to the conclusion that:

1. one state can have a transition to itself, which is called a self transition.
2. one transition must have an event, but does not have to have an action.

Now we will explain the State Chart in detail for the next Algorithm, namely protocol "All the way". For now, we will just explain the algorithm and the State Chart individually without comparing with the previous one.

Recall LeLann's algorithm overviewed in the second chapter. The State Chart is in Figure 5.3. Now we are going to explain this algorithm's State Chart. The nodes participate in the leader election by running protocol "All the way". Initially, they are all in the initial state "Asleep". When a node is waken up by a timer, it typically originates a message containing its own ID to one of its neighbors if this node is an initiator, not a relay node (see the definitions in Chapter 2). If it is a relay node, it does nothing. This is captured in the State Chart using the two transitions started from state Asleep to the state Candidate and to the state Relay respectively. On the first transition, the event "Initiator wakes up (time out)" is followed by action "Originate electing message". On the second, when the event "Relay node wake up (time out)" happens in state Asleep, then we go to state Relay, but perform no action.

We now adopt the viewpoint of a single node again to explain the algorithm and its State Chart. If a node is an initiator and is in Candidate state, there are two events that can make this node go into two other states, and there is one other event that will trigger a self transition from Candidate state to itself. The self transition is triggered by event "Received an electing message from the other node, which is not the last one" (i.e., it is not the last message to receive by this node). "Record the message and pass this message" is the action that follows this trigger. This transition means: if a node in the candidate state (i.e., the node originated a message) receives a message that was not originated by itself, and the number of this message (in the count kept by this node) is less than the number of nodes in the ring (or the number of initiators, if there is any relay node), then the node records the ID in the message and passes the message. But the node will remain in the state Candidate. The idea is that each initiator node accumulates the IDs of all other initiators.

Let us consider the other two transitions that from state Candidate to the other two states. First, the event "Received all the electing messages from the nodes in the ring, and itself is the smallest" triggers the transition going to state Leader and followed by no action. This captures the fact that only a node that fulfills the following three requirements can become the leader: 1) this node once was an initiator, 2) it received all the messages originated in the ring 3) it confirmed that it has the

smallest ID of all nodes in the ring.

The last transition starts from state Candidate goes to state Follower. This transition is triggered by event "Received all the electing messages from the nodes in the ring, and itself is not the smallest". There is no other action after the node becomes a follower. This fact illustrate that each node in the ring will realize by itself whether it is a leader or a follower once it receives all the messages originated in the ring. Each node will compare all the IDs and figures out who has the smallest ID. If it has the smallest ID in the ring, then it becomes a leader; otherwise it become a follower. The transitions related to state Relay are similar to some of the transitions on state Candidate. The only difference is that if a node is a relay node then it cannot become leader. This functionality is related to feature "Election Scope" in our feature model. We will explain the feature model and state machine relationship later.

5.4 Integrating the first two algorithms in Rational Rose RT.

In the previous section we built two state machines for protocol "All the way" and Protocol "AsFar". So far, we haven't related these algorithms with features, except

Figure 5.3: The State Chart for Protocol "All the way" (LeLann's algorithm)

	Protocol “All the way” (Lelann’s)	Protocol “AsFar” (Chang and Robert’s)
Feature Values	F1V1 Unidirectional Ring (or F1V2 Bidirectional Ring) F2V1 Oriented Ring F3V1 Among all nodes F4V1 Elect minimum F5V1 Single stage F6V1 Randomly choose direction F7V1 Complete F11V2Do not Use Notify Message F12V1Same Behavior	F1V1 Unidirectional Ring (or F1V2 Bidirectional Ring) F2V1 Oriented Ring F3V1 Among all nodes F4V1 Elect minimum F5V1 Single stage F6V1 Randomly choose direction F7V2 Block F11V1Use Notify Message F12V1Same Behavior

Figure 5.4: The Feature value - Algorithm Corresponding diagram for protocol “All the way” and protocol “AsFar”)

for the Feature Algorithm Relationship offered as a summary in chapter 2.

Let’s take a quick look at which feature values do protocol “All the way” and protocol “AsFar” correspond to (See Figure 5.4).

From the table above we can highlight some of the differences and similarities between those two algorithms: they can both run in Unidirectional Ring (F1V1) and Bidirectional Ring (F1V2); they both need to have the sense of direction (in other words the direction “right” should be globally consistent (F2V1 Oriented Ring)); they are both algorithms that elect a leader from all the entities in the network

(F3V1 Among all nodes); they both elect the node that has the smallest value to be the leader (F4V1 Elect Minimum); they both can finish the election within one stage, namely they only originate message once (F5V1 Single stage); when they are ready to send messages, they can send to the right or left, both directions will be the same because the ring topology is totally symmetric (F6V1 Randomly choose direction); in either algorithm, each node will behave the same throughout the election (F12V1 Same Behavior).

When considering the integration of these two algorithms, it is the integration of their differences that is more problematic. The difference between these two algorithms are:

- 1) a message in the protocol "All the way" will travel along the ring until it comes back to its originator (F7V1 Complete), but the messages in the protocol "AsFar" will only travel until it meets a node with a smaller ID (F7V2 Block);
- 2) in protocol "AsFar", after the leader is elected, the system needs the leader to send a notify message to notify all the other nodes, to let them become followers (F11V1 Use Notify Message); but in protocol "All the way" each node will get all the messages that originated in the ring network. Each node will compare all the IDs contained in the messages, then figure out who has the smallest value and whether itself has the smallest ID value. All the nodes will eventually know who is the leader and who is a follower. So, no need for sending a notify message (F11V2

Do not Use Notify Message).

In the individual algorithm State Charts we described in the previous section, we did not use any feature to describe the events on the transitions. Now we will integrate the two state machines and use choice of feature value to express the differences between two algorithms.

In Figure 5.2 there are two final states: Terminate and Follower. We found that in Figure 5.3, the two final states are: Leader and Follower. That is affected directly by feature "Notification". When the algorithm needs to use a notify message to inform the follower and terminate the system, the state Terminate will be needed. If an algorithm does not need to use a notify message to terminate the system, then after each node looks at all its messages received and finds out who is the leader and who are the followers, then the Leader and Follower states act as final states. In order to integrate the two State Charts, we need to keep the Leader, Terminate and Follower states in the integrated State Chart: conceptually these are indeed the same states. And in order to make the final states be consistent, we combine the states Leader and Follower into a single state we call Terminate. Access to this state can be controlled by the trigger events on transitions.

As mentioned in the second section of this chapter, Rational Rose RT provides

Figure 5.5: The integrated state machine for two algorithms - Event code on transition from state Candidate to state Terminate

an executable State diagram. That is, we can write C++ code on each transition to control the executable state diagram. Figure 5.5 is the event code (see the highlighted part) on the transition from state Candidate to state Terminate. Figure 5.6 is the action code (see the highlight part) on the transition from state Candidate to state Terminate.

Now let's address the second difference between these two Algorithms, namely the problem of how far should a message go. See Figure 5.4: protocol "All the way" will have the messages go along and complete the entire ring. Protocol "As-

Figure 5.6: The integrated state machine for two algorithms - Action code on transition from state Candidate to state Terminate

Figure 5.7: The integrated state machine for two algorithms - Action code on self transition on state Candidate

Far” will ask each node to check whether the ID contained in the message it receives is larger or smaller than its own. If, we repeat, the ID from the message is larger than its own ID then, the message will be blocked, otherwise, the node will pass the message along the ring. Let’s give the corresponding event code in Figure5.7 and action code in Figure5.8

5.5 Integrating ”Single Stage” and ”Multiple Stage”

In the previous section we integrated two algorithms that will successfully elect a leader using only one stage. Those two are the easiest two to integrate! When we integrated the group of algorithms with multiple stages, we faced many more

General | Triggers | Actions | Files |

Code:

```

int i;
WakeUpData *wd = (WakeUpData*)rtdata;
RTProtocol *inport= msg->sap();

if ((wd->ID != MyID) && (NOTIFICATION == NoNotifyMessage) && (NumberOfNodes != NumberOfNodesR
{
    if (NumberOfNodes != NumberOfNodesReceived)
    {
        NumberOfNodesReceived++;
        wd->NodesCounter++;
        if((wd->ID < SmallestID) || (SmallestID==0))
        {
            SmallestID = wd->ID;
            printf("\n\n*** node %i, the smallest id is %i, got message %i\n\n", MyID,SmallestID, w
            left.wakeUp(wd).send();
        }
        else
        {
            Nodetempl.done().send();
        }
    }
}

if ((wd->ID != MyID) && (NOTIFICATION == NeedNotifyMessage))
{
    if(STAGES == Single_Stage)
    {
        printf ("\n\n***my id= %i, smallest id= %i\n\n",MyID, SmallestID);
        if((wd->ID > SmallestID) && (SmallestID!=0))
        {
            printf("\n\nmy id %i, smallest id %i, the bigger id I received %i\n",MyID,SmallestID
            delete wd;
        }
        else if((wd->ID < SmallestID) || (SmallestID==0))
        {
            printf("\n\nmy id %i, smallest id %i, the smaller id I received %i\n",MyID,Smallest
            SmallestID=wd->ID;
            left.wakeUp(wd).send();
        }
    }
}

```

Browse ▾

OK Cancel

Figure 5.8: The integrated state machine for two algorithms - Action code on self transition on state Candidate to state Terminate

differences between the different algorithms. Also, with the increasing number of variation points, the integration is harder and harder, and the feature model and our novel FSDF Chart play more and more important roles.

While we cannot explain at length every decision taken during this massive integration, we can highlight a crucial one. Here we will show some details of how we integrated single and multi-stage algorithms. Then, in the next section, we will briefly talk about the stated-based testing.

More than eighty percent of leader election algorithms in a ring are using a multiple stage strategy to elect the leader. Santoro explains: multiple stage leader election in ring algorithms have guaranteed $O(n \log n)$ message performance, whereas single stage algorithms have $O(n^2)$ message performance[41]. In multiple stages, the message originator can control the messages even after it sends them out. If an algorithm uses multiple stages, then the entities in the network will not obey a single behavior any more; they will change behavior according to the result of each stage or sub-step in each stage (in the case of most unidirectional ring multiple stage algorithms – See feature combination rule in chapter 3). In feature "Terminate Passing Message Strategy", there are four choices all together: those four choices are the most important fact to affect the way each algorithm controls its messages. From Figure 5.9 we can see how we use the choices of feature "Terminated Passing Mes-

Figure 5.9: The integrated state machine – Action code on self transition on state Candidate to state Terminate
 sage Strategy”.

In our integration, we need to think of the flow in the whole State Chart once we have an idea of which feature(s) control(s) which facets of the algorithms in the domain. For this purpose, we introduced the FSDF Chart in chapter 4. In Figure 5.10 we combined the Flow Chart and decision tree of our domain, used features as the decision points and feature values as the decision branches, and used states of the integrated state machine as the nodes in the graph. We also used the type of the messages received to control the flow. In summary, this diagram not only

Figure 5.10: The integrated FSDF Chart

makes clear the entire flow of the integrated domain, but also bridges features and their values to the states of the target State Chart: it is thus pivotal in the integration process.

Having bridged between features and states using the FSDF Chart, we can also finalize the flow of execution in the integrated algorithm. The last step is to enter our model into Rational Rose RT to get the model to execute. Recall that we summarized the combination rules for feature interactions in chapter 3. In Figure 5.11 we show how to express the feature combination rules in our Rose RT model. More specifically, we define the combination rules in the Verify function of the transition

Figure 5.11: Capturing Feature Combination Rules in the top level state diagram in integrated model

VerifyFeatureData. Note that contrary to traditional state machines, the state model in Rose RT requires that the user name all the transitions, states, decision points and the other components. Here VerifyFeatureData is the name for decision point in the top level state diagram in our model. This is another important observation: we use a hierarchical state machine in order to avoid clutter of states and transitions. Such hierarchical state machines also have the advantage of allowing the designer to separate concerns in distinct hierarchical states (i.e., states that themselves contain a state machine).

In the final integrated state machine (See Figure 5.12), we have five states: Can-

didate, Defeated, Leader, Follower and Terminate. Recall the integrated state machine for the two single stage algorithms has an other state called Relay. But in multiple stage algorithms, after each stage, the entities will have to decide: keep participating in the next stage or give up the chance of being elected in the next stage (here called defeated). This decision will be based on the results of current stage. If an entity still has a chance of being elected in the next stage, then it stays in Candidate state, otherwise it does not originate a message any longer and goes to Defeated state immediately. If an entity is defeated, then it only passes the messages it receives without doing any analysis of the message. Hence, the state Defeated has the same function as state Relay. So, we combine state Relay and state Defeated into one. We will also call this state Defeated.

The most interesting part of Rational Rose RT is that it can actually run the state machine we just build by showing the transition of the messages among states with high quality graphic interface. The following Figure 5.13 shows the state machine of one node running in the network. The graph recorded the situation of one node changing from state Candidate to state Follower. This transition was triggered by event "got a Notify Message".

Figure 5.12: Second level state diagram of integrated executable model

Figure 5.13: A Running State Machine Model on One of the Node in the Ring

Chapter 6

Conclusion

6.1 About our Contributions

To the best of our knowledge this is the first time a feature model and its corresponding executable model have been developed for a family of distributed algorithms. We believe this work offers two significant contributions.

First, in the area of system family engineering, it represents an interesting application of the ideas of that body of research. Most work so far in this area has focused on implementation techniques (especially in C++ and Prolog), and feature models and their generators for libraries of data structures (such as matrices). We believe our work demonstrates the feasibility of applying domain engineering ideas to families of algorithms. Also, choosing to use ROSE Real Time opens the door

to the possibility of having test generation across the domain. Recall that Binder [11] explains at length how to test state machines: this is a promising area for future work. And the idea of using a hierarchical state machine in lieu of a generator also has the merits of providing a new approach to generation: features can be used to control the selection of behavior (across family members) at run time. Finally, our proposal to introduce a new model to bridge from features to states addresses one of the main difficulties of system family engineering, namely, the bridge from the problem space to the solution space.

Second, from the viewpoint of research on algorithms, we believe our work offers researchers with a method for algorithm conceptualization, integration and simulation. With respect to algorithm conceptualization, clearly system family engineering emphasizes variability within a family, a viewpoint which it exploits to its fullest in order to produce an integrated conceptualization and implementation of such a family. That is, thinking about variability leads us to invest in a unified conceptual framework, which itself can be used to produce an integrated model. Having the latter be executable by an industrial-strength tool offers yet another advantage: independent researchers can readily reuse such work since ROSE-RT offers standardized semantics through UML. In other words, what can be expressed and how the model is executed are precisely defined.

In hindsight, we believe this work may also help researchers in exploring new algorithms (viewed as new configurations of features within the domain): the more one knows about the variation points in the domain, the likelier it is one will come up with new features combinations and develop new variations. This is a promising way to discover new algorithms and possibly improve the existent algorithms. Moreover, one will also be able to reuse parts of existing algorithms.

6.2 Future research in the context of this thesis

As we conclude this thesis, we find there are several promising directions for future work we need to explore as a research group. With respect to the family of algorithms we investigated, clearly we would like to first invest a considerable amount of time in the testing of the model. We have tested each of the algorithms we have integrated to verify basic functionality. More testing is required as well as studying the testing tools that come with ROSE-RT to see if they could be of help. In the same vein, we would like to explore how to realize some of the ideas puts forth for the systematic testing of state machines. We also need to systematically verify feature interactions and, most interestingly, to explore new combinations and study the behavior they generate. In other words, we would like to see how to systematically scrutinize the behavior of 'new' family members.

With respect to the whole approach, other students in our research groups are formalizing feature interactions and tying features to non-functional requirements. Lots remains do be done here. In particular, can we decentralize the verification of feature interactions? And how can we associate non-functional requirements to configurations of features?

Finally, clearly, we hope that others will have the opportunity to apply the ideas of this dissertation to other families of algorithms. State-based behavior is obviously the first ingredient to look for in such families if one wants to maximize the reuse of our contributions.

Bibliography

- [1] Customer service process flowchart ([http : //www.business – process – management – software.com/customer – service – process – flowchart.asp](http://www.business-process-management-software.com/customer-service-process-flowchart.asp)).
- [2] Decision tree induction ([http : //www.cs.bris.ac.uk/ flach/presentations/idahtml/sld006.htm](http://www.cs.bris.ac.uk/flach/presentations/idahtml/sld006.htm)).
- [3] Finite state machine ([http : //en.wikipedia.org/wiki/e](http://en.wikipedia.org/wiki/e)).
- [4] Finite state machine ([http : //www.nist.gov/dads/html/finitestatemachine.html](http://www.nist.gov/dads/html/finitestatemachine.html)).
- [5] Flowchart - a whatis definition ([http : //whatis.techtarget.com/definition/0,,sid9_gci212134,00.html](http://whatis.techtarget.com/definition/0,,sid9_gci212134,00.html)).
- [6] Incremental decision tree induction ([http : //www.cs.umass.edu/ lrn/iti/dtree – background.html](http://www.cs.umass.edu/lrn/iti/dtree-background.html)).
- [7] Edward Yourdon . *Death March: The Complete Software Developer's Guide to Surviving 'Mission Impossible' Projects*. Prentice Hall,, 1997.

- [8] J. E. Hopcroft A. V. Aho and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company,, 1974.
- [9] Michael A. Arbib, A.J. Kfoury, and Robert N. Moll. *Introduction to Formal Language Theory*. Springer Verlag, New York, New York, USA,, 1998.
- [10] D. Batory and S. OMalley. The design and implementation of hierarchical software systems with reusable components. *In ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 4, pp. 355-398., October 1992.
- [11] Robert V. Binder. *Testing object-oriented systems – Models, Patterns, and Tools*. Reading, MA: Addison-Wesley,, 1999.
- [12] Ernest J. H. Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the A.C.M.*, 22(5):281-283, May, 1979.
- [13] Krzysztof Czarnecki. Generative programming and software system families .
- [14] H. DeBruin and H. van Vliet. *Quality-Driven Software Architecture Composition*,. ECBS,, 2002.
- [15] Bruce Powel Douglass. Statecharts: a visual formalism for complex systems,. 1998.

- [16] Rational Rose @ RealTime Evaluation Edition. Rational rose rt online help. 2002.
- [17] Doron Rotem Ephraim Korach and Nicola Santoro. Distributed election in a circle without a global sense of orientation. *Int. J. of Computer Mathematics*, 16:115-124,, 1984.
- [18] M. Fowler. *Analysis patterns: reusable object models*. Addison-Wesley,, 1997.
- [19] W.R. Franklin. On an improved algorithm for decentralized extrema-finding in a circularconfiguration of processes. *Communications of the A.C.M.*, 25(5):336-337,, May 1982.
- [20] P K. Srimani G. Antonoiu. A self-stabilizing leader election algorithms for tree graph. *Journal of Parallel and Distributed Computing.*, 2003.
- [21] David. Harel. Real-time uml: Developing efficient objects for embedded systems . *Science of Computer Programming*. Vol. 8, p. 231., 1987.
- [22] Derek J. Hatley and Imtiaz A. Pirbhai. *Strategies for real-time system specification*. New York: Dorset House, 1987.
- [23] L. Higham and T. Przytycka. A simple, efficient algorithm for maximum finding on rings. *Information Processing Letters*, 58:319-324,, 1996.

- [24] D.S. Hirschberg and J.B. Sinclair. Decentralized extrema-finding in circular configurations of processors. *Communications of the A.C.M.*, 23:627-628,, 1980.
- [25] John E. Hopcroft and Jefferey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Adison-Wesley Publishing Company, Reading, Massachusets, USA,, 1979.
- [26] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to automata theory, Language, adn computation*. Reading, Mass.: Addison-Wesley,, 1979.
- [27] Z. Wu J.-P. Corriveau, T. McClean1. Envisioning domain-driven validation. *TOOLS-USA*,.
- [28] D. Rotem J. Pachl and E. Korach. Lower bounds for distributed maximum finding algorithms. *Journal of the ACM*, 31:905-917,, 1984.
- [29] Jacobson. Oose and book on reuse. 1999-2000.
- [30] Toby McClean1 Jean-Pierre Corriveau. Scenario-driven quality engineering in domain modeling. 2003.
- [31] J. A. Hess W. E. Novak K. C. Kang, S. G. Cohen and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study.
- [32] Ulrich W. Eisenecker K. Czarnrcki. *Generative Programming - Methods, Tools, and Applications*. Addison Wesley, 2002.

- [33] B.L. Kovitz. *Practical Software Requirements*. Manning,, 1999.
- [34] Gerard LeLann. Distributed systems - towards a formal approach. 1977.
- [35] S. Tek M. Madigan. Requirements verification and management,.
- [36] Carma McClure. Model-driven software reuse practicing reuse information engineering style. *Extended Intelligence, Inc*, 1995.
- [37] J. Neighbors. Software construction using components. *Ph. D. Thesis, (Technical Report TR-160), University of California, Irvine,, 1980*.
- [38] G.L. Peterson. An $o(n \log n)$ unidirectional algorithm for the circular extrema problem. *A.C.M. Transactions on programming languages and systems*, 4(4):758-762,, oct 1982.
- [39] Emmanuel Roche and Yves Schabes. *Finite-State Language Processing*. Bradford Book series, MIT Press, Cambridge, Massachusetts, USA,, 1997.
- [40] G. Prencipe N. Santoro S. Dobrev, P. Flocchini. Multiple agents rendezvous in a ring in spite of a black hole. *Extended Intelligence, Inc*, 2003.
- [41] N. Santoro. *Design and Analysis of Distributed Algorithms*. unpublished, Manuscript, 2002.
- [42] W. ed. Tracz. Software reuse: Emerging technology. *IEEE Press*, 1988.
- [43] Paul T. Ward and Stephen J. Mellor. *Structured development for real-time systems*. Englewood Cliffs, N.J.: Prentice-Hall, 1985.

[44] Roger Wattenhofer. *Principles of Distributed Computing*. Summer 2003.

[45] J. Withey. Investment analysis of software assets for product lines. *Technical Report CMU/SEI-96-TR-010, Software Engineering Institute*, 1996.