INFORMATION TO USERS

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

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

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

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

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

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

DERIVING NEW MEASUREMENTS FOR REAL-TIME REACTIVE SYSTEMS

Olga Ormandjieva

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Doctor of Philosophy
Concordia University
Montréal, Québec, Canada

April 2002

Abstract

Deriving New Measurements for Real-Time Reactive Systems

Olga Ormandjieva, Ph.D.
Concordia University, 2002

Real-time reactive systems are largely event-driven, interact intensively and continuously with the environment through stimulus-response behavior, and are regulated by strict timing constraints. Examples of such systems include alarm systems, air traffic control systems, nuclear reactor control systems and telecommunication systems; applications involving real-time reactive software play a mission-critical role in the defense industry.

Real-time reactive systems are inherently complex. The complexity pervades through the different phases of software development, deployment, and maintenance. Applying formal methods in the development process is an effective way for dealing with the complexity, and for quality assurance. One of the goals is to assess the quality of such systems starting from the earlier phases of their life cycle.

The integration of the quality measurement into the development framework provides feedback to the system developers in order to effectively control the development processes and to obtain high reliability of a final product. Thus, quality control is a must when safety-critical real-time reactive systems are developed. The quality assessment must be regarded as a support for controlling the process of software development in order to guarantee the final quality.

The aim of the thesis is to correctly apply the measurement theory to formal description of real-time software upon which we can base models of object-oriented software measurement. In order to create the framework for the present work, we are surveying the theoretical approaches to software measurement.

The novelties of the quality measurement methodology are in the theoretical basis and a practical automated measurement data generation process for real-time reactive systems. The proposed approach is applicable to real-time reactive systems modeled as timed labeled transition systems.

To my family.

Acknowledgments

I would like to thank my supervisor, Dr. V.S. Alagar who guided this work with good advice, constant encouragement and insightful comments, and provided continuous support throughout my Ph.D studies.

Also, I would like to thank all Faculty and Staff, and the TROMLAB research group for their support during these years.

On a personal level, I thank my husband Chris for his support and help, and my three wonderful daughters for their patience, understanding and love.

Contents

List of Figures

Chapter 1

Introduction

1.1 Software Measurement

The goal of *software engineering* is to apply an engineering approach to the construction and support of software products so they can safely fill the uses to which they may be subjected. As any engineering approach, software engineering requires a measurement mechanism to provide feedback and assist the software development, testing and maintenance.
Informally we can define measurement as *a process of quantifying the attributes of software in order to characterize them according to clearly defined rules.*
Like any engineering activity, software measurement requires a definition of the environment in which the measurement is expected to be performed. The knowledge on what is measurable, when it should be measured, and well-defined measurement purpose(s) are necessary.

1.1.1 Who you measure for?

Fenton et al. [FP97] define three categories of software entities we can measure: products, processes and resources. *Products* are the deliverables created during the course of a project (for example, requirements, functional specifications, design documentation, source code, test cases, test results, etc.). *Processes* are the set of activities used by an organization to develop its products. *Resources* are the input to the process used on a project (i.e., people, tools, materials, methods, time, money, products from other projects). Whitemire [W97] has added one more category, the

project, defined as the relationship between instances of processes, products, resources and internal/external goals, standards and constraints.

1.1.2 What you measure?

Fenton et al. [FP97] classify the measurable characteristics of the software entities as internal and/or external. *Internal characteristics* are those attributes of software entities which can be measured purely in terms of the process, product, or resource itself. One application of internal metrics is in terms of a threshold value or "alarm" [HS96]. Such a value depends on the particular development environment, especially the complexity of a problem itself.
External characteristics are those attributes which can only be measured with respect to how the process, product, or resource relates to its environment. For example, external product attributes include quality, reliability, testability, reusability, and maintainability. Internal product attributes include size, complexity, reuse, defects, coupling, cohesion and polymorphism.

1.1.3 Why you should measure?

There are two general applications of software measures: *evaluation*, used to assess an existing software entity by numerically characterizing one or more of its attributes: and *prediction*, used to predict some attribute of a future software entity, involving a mathematical model with associated prediction procedures. Whitemire [W97] gives more detailed meaning to the evaluation subdividing it in *estimation, assessment, comparison and investigation*. The prediction measurement may be applied from within the early phases of software development to predict future characteristics of software entities.

1.1.4 When you should measure?

The essential goal of software measurement is to identify an anomaly within the same development phase in which it originated, as well as to measure development progress. Thus, each software development phase should contain metrics in order to achieve high project visibility and control on quality.

1.1.5 What measures to use? GQM (*or* FCM) Approach

The *Goal-Question-Metric (GQM) paradigm* is a common approach to set the measurement framework [FP97], also known as *Factor-Criteria-Metric (FCM) paradigm*. The GQM approach is a six-steps method for measuring within the context of an organization or specific project. You need a set of metrics necessary to assess the achievement of who, why and when you measure, to collect meaningful measurement data and to analyze it according to clearly defined rules.

Step 1. *(Conceptual Level) Develop a set of* **goals (factors).**

A **goal** is a five-tuple *Goal=(purpose, issue, object, viewpoint, environment)*

Example of use: *Goal=(improve, timeliness, change request, project manager, development process).*

Step 2. *(Operational Level) Develop an operational model.*

A set of **questions (criteria)** is developed to verify or assess the achievement of a particular goal.

Step 3. *(Quantitative Level) Determine the measures needed.*

A set of **theoretically valid measures** is associated with each question in order to answer it in a quantitative way.

Step 4. *Develop a mechanism (tool) to collect and analyze the data.*

Step 5. *Collect measurement data, and* **empirically validate the measures.**

Step 6. *Analyze the data and feed back to the projects.*

The *numerical observations* are converted into *numerical results* using any of several statistical and mathematical techniques. Once interpreted into *empirically valid results*, we can use this information for the measurement purpose we had in mind.

1.1.6 Properties of Software Measures

The different views on desirable software measurement properties are summarized in [HS96]. From the practical point of view, the measure must be economical to collect

and automate, easy to apply and calculate, relevant to the user of the measure, and (when appropriate) obtainable in early life-cycle phases.

A software measure in general has to be objective, reliable, valid, and robust. *Objectivity* means that the measurement process should not depend on the subject (person, tool) that performs the measurement, on system's size, or on the programming language used (in case of code metrics). The *reliability* requires the metrics to characterize in a unique way every entity measured. Equal entities should obtain equivalent measurement values, repeated measurement in equal conditions should give same values for the same entities. The *robustness* addresses the ability of a measure to tolerate incomplete information. The software measure requirement *validity* means that the measurement data should reflect exactly the characteristics of the entity under measurement.

There are three different types of validity: *face* (intuitively relevant to the user); *internal* and *external*.

Internal validity addresses how well a measure captures real differences in the values of an attribute of the real-world entities being measured. Internal validity can be *content-related* (the coverage that the measure provides of the attribute), *criterion related* (the level of accuracy of prediction or estimation), or *construct-related*. The last one deals with the theoretical validation of a new measure, experimental *convergent validation* with previously validated measure(s) of the same attribute, and the *discriminant validation* to prove the new measure has a low correlation to validated measures of attributes unrelated to the given one.

The existing results of research on software measurement reported in the literature show that universally validated and tested metrics do not currently exist, especially for OO environments. Most of the publications are related to the conventional software and object-oriented software. The described empirical metrics are locally useful in the context of alarm triggers, but are highly parochial, highly limited, and highly unscientific [HS96].

1.2 Real-Time Reactive Systems

Real-time reactive systems are largely event-driven, interact intensively and continuously with the environment through stimulus-response behavior, and are regulated by

strict timing constraints. In other words, real-time reactive systems are in constant relationship with their environment and the stimulus-response behavior respects time constraints that ensures its correct and safe operations. The term *reactive* was introduced by Harel and Pnueli [HP85] to designate systems that continuously interact with their environment and to distinguish them from the *interactive* and *transformational* systems. Two important properties distinguish the reactive systems from other real-time systems:

- *stimulus synchronization:* the process always reacts to a stimulus from the environment;
- *response synchronization:* the time elapsed between a stimulus and its response is acceptable to the relative dynamics of the environment, so that the environment is still receptive to the response.

Examples of such systems include alarm systems, air traffic control systems, nuclear reactor control systems and telecommunication systems; applications involving real-time reactive software play a mission-critical role in the defense industry.
The main issue in the development of safety-critical systems is to produce a reliable design. The real-time system design is a conceptual solution to the domain problem and is the basis for an implementation of the solution. Its quality is essential for the economics of the software development and the reliability of the final product. Real-time reactive systems are inherently complex. The complexity pervades through the different phases of software development, deployment, and maintenance. The factors that contribute to real-time systems complexity include time constraints on stimuli and responses, safety requirements, complex sequencing of events, and concurrency. To achieve a high level of reliability, the design must be supported by a rigorous formalism. The formal object-oriented method TROM [Ach95] has been studied as a formal basis for the development of real-time reactive systems. The TROM formalism is founded on merging object-oriented and real-time technologies, and provides a formal basis for specification, analysis and refinements of the real-time reactive systems design.

TROMLAB [AAM98] is a framework for real-time reactive systems development built on TROM formalism. The framework includes a number of tools to promote a rigorous development of real-time reactive systems. As the measurement procedure

is both time and resource consuming procedure, a tool for automatic gathering of quality measurement data and analyzing it according to the quality requirements, has been designed.

1.3 Research Goals

In the context of real-time reactive systems, which are mostly safety-critical, the main motivation for quality measurement comes from the requirements for correct implementation of safety and time-dependent behavior. Quality assessment must be conducted right from the design specification stage to ensure correct performance.
This thesis proposes a quality assessment based on measurement in real-time reactive systems in different phases of their life-cycle. The proposed measurement model is developed in the context of the TROM formalism. The TROMLAB is a framework for applying the TROM method and incorporates rigorous methods for validation, verification and refinements of the design.
The proposed approach is applicable to real-time reactive systems modeled as timed labeled transition systems, and developed according to the process model shown in Figure 1.
The process model shown in Figure 1 has five different phases: requirements analysis and specification, design, validation and verification, implementation and testing. The analysis phase defines a domain problem (real-world objects, associated processing and timing requirements), and constructs class specifications. The design constructs the subsystem from objects of the specified classes and is a conceptual solution to the domain problem. It forms a basis for validation, verification, reasoning, testing, and an implementation of the solution. Measurement has to be incorporated into the specification, design, implementation and testing phases of the above process model in order to assess quantitatively the quality of the final product. The virtues of the process model include iterative development, incremental design, and application of formalism through the different stages of development.

1.3.1 Quality Measurement Model in Real-Time Systems

The integration of the quality measurement into the development framework provides feedback to the system developers in order to effectively control the development

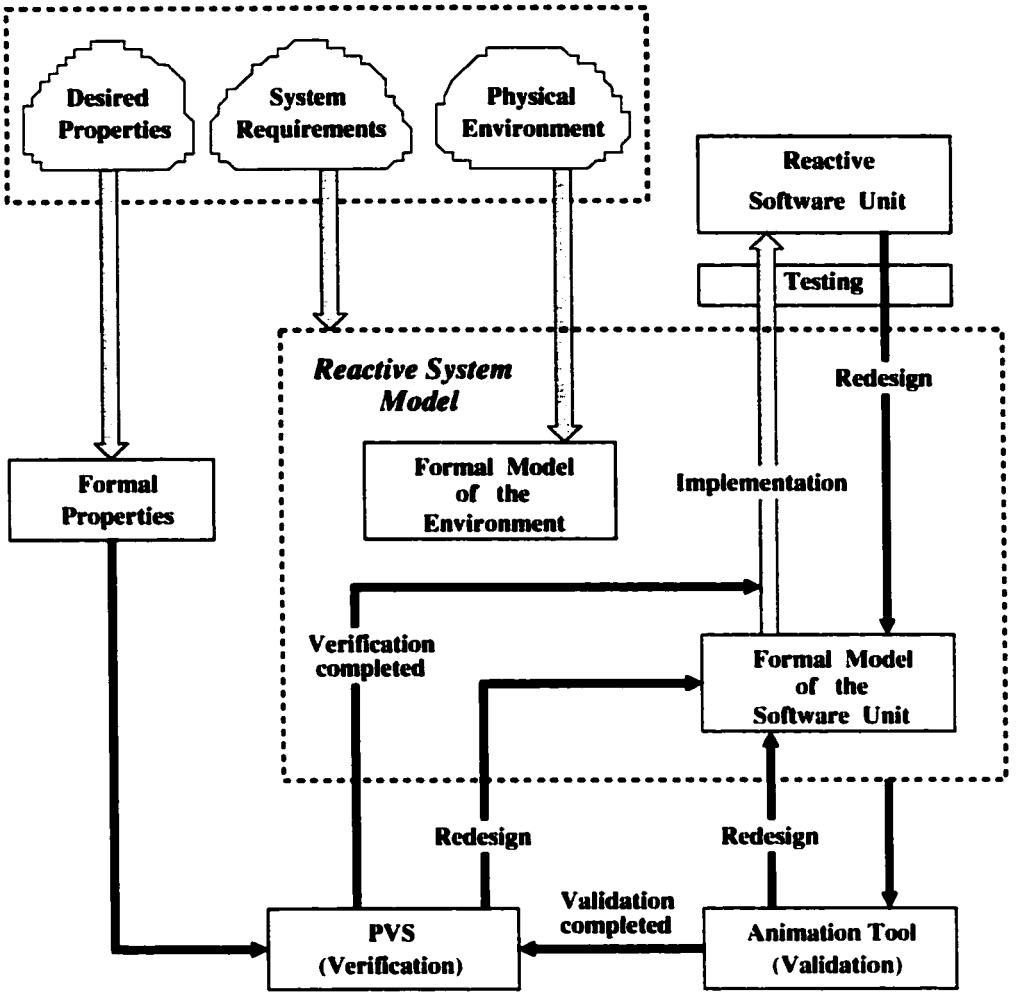


Figure 1: Process model for developing complex reactive systems

processes and to obtain high reliability of a final product. Thus, quality control is a must when safety-critical real-time reactive systems are developed. The quality assessment must be regarded as a support for controlling the process of software development in order to guarantee the final quality.

One of the main issues, when measuring real-time reactive systems, is the reliability of the measurement process itself. To achieve this, the quality measures are based on the theory of software measurement, and the components and development processes to be measured are precisely described on the basis of a rigorous formalism - TROM.

1.3.2 Formal Approach to Measurement Validation

Any measure should have a firm grounding in both the empirical understanding of the software attribute to be measured, and the measurement theory. The foundations of the software measurement methodology rest on the representational measurement theory [FP97]. The theoretical validation of the measure guarantees the correctness of the measurement data collection process in a specific environment.
In most disciplines, measure validation happens in carefully controlled experiments. In software development and maintenance, controlled experiments are resource consuming and that increments the cost of measurement process. However, for most of the measurement purposes, it is sufficient to formally prove that the measure meets the representation condition, and to show empirically that the values of the measurement data correspond to the expected results.
The aim of the thesis is to establish the basics of a theory of measurement for the real-time software development, and to give the theoretical tool for building measures from the theory out (and thus theoretically validated a priory). This approach would reduce the expensive empirical validation in industrial environment to a few experiments. The approach has been applied to the measures developed for the quality assessment in the TROMLAB environment.

1.4 Major Contributions and Thesis Outline

In this thesis, a theoretical framework is laid for metrics and measurements of real time reactive systems and their practical applicability is illustrated with a case study. The research results include methods for measuring architectural complexity, maintainability, testability, test adequacy, and reliability. The theory is developed for real-time reactive system developed in TROMLAB framework. However, it is applicable for assessing the quality of real-time reactive systems whose behavior can be abstracted as timed labeled transition systems.

The two inherent properties of a reactive system are *stimulus synchronization*, and *response synchronization*. In a real-time reactive systems, strict time constraints govern response times as well as internal computations. A critical study of some industrial systems, such as Nuclear Power Plant Control System, reveal that real-time

reactive systems also involve concurrency in addition to time-constrained synchronization. Based on such investigation we have proposed a hierarchical quality model for real-time reactive systems. The factors, criteria, and measures identified in this model are investigated formally to choose appropriate metrics and derive new measures for real-time reactive systems.

Chapter 2 reviews **TROMLAB**, an environment for developing real-time reactive systems according to **TROM** formalism and gives the context of research described in this thesis. The *Train-Gate-Controller* example, a bench mark case study in the real-time systems community, is introduced, with a complete description of its visual models and formal specifications. Chapter 3 is a brief survey of formal approaches to software measurement. Measurement models and measures of software quality that have a bearing on the research directions pursued in this thesis are reviewed in this survey. Chapter 4 introduces a quality assessment model for real-time reactive systems. It is based on the *Software Quality Factor-Criteria-Metrics Framework* [IEEE93]. The goal is to develop the criteria and measurements that are appropriate for our work. Chapter 5 contains several major results:

- Architectural Complexity: A mathematical model of the metric is introduced. Based on that metric, architectural complexity and local architectural complexity measures are developed.

- Maintainability Measure: This metric is based on the architectural slice extraction method applied to the system architecture.

- Testability Measurement: This measure is related to the input/output behavior of a reactive object. An information-theoretic measurement is developed to quantify the controllability of an object's input and the observability of the object's output.

- Functionality Measure: The functional complexity measure applies to an implementation of the system, as derived from the architectural design.

Chapter 6 rigorously defines test adequacy, provides a formal representation of test domain, introduces a mathematically valid metric, and gives an algorithm to select test cases based on the metric. Chapter 7 introduces a Markov model of a reactive system and provides a formal approach to compute reliability prediction of an evolving

real-time reactive system. The thesis is concluded in Chapter 8 with a summary of major contributions and future research directions.

Chapter 2

Background

The quality assessment is discussed in the context of TROMLAB, an environment for rigorous development of real-time reactive systems built according to the process model shown in Figure 1. The goal of this chapter is to describe the formal object-oriented formalism (TROM) and the TROMLAB framework for which the theoretically valid quality measurement model is developed.

2.1 Development Methodology

The TROM formalism [Ach95] is founded on merging object-oriented and real-time technologies, and provides a formal basis for specification, analysis and refinements of the real-time reactive systems.

2.1.1 Abstract Reactive Models

A reactive object is modeled abstractly as a finite state machine augmented with ports, attributes, logical assertions on the attributes, and timing constraints. To manage complexity, we distinguish between a simple state, and a complex state, such that a complex state is an encapsulation of another finite state machine, with an initial state, and which can include other complex states. System objects communicate using a synchronous message passing mechanism. An external event, either input or output, can only occur at an instance of a specific port type; an internal event occurs at the null port. Thus a port type symbolizes the events that can occur through its instances; events label the transitions between states. The type of an attribute can be either a

port type, or an abstract data structure modeled as an LSL (Larch Shared Language) [GH93] trait. Logical assertions on the attributes specify a port condition, an enabling condition, and a post condition on each transition. Timing constraints are associated with a transition to describe the time-constrained response to a stimulus. A generic reactive class (GRC) having port types, attributes, logical assertions on the attributes and time constraints models a collection of reactive objects.

An abstract model of a subsystem includes instances of generic reactive classes, each with instances of each port type of the corresponding class. Two ports are compatible if the set of input message sequences at one port is equal to the output message sequences at the other port. A port link connects two compatible ports. A port link is an abstraction of communication mechanism between the objects associated with the ports connected by the link. The port links effectively determine the set of all valid messages that can be exchanged among the objects in a subsystem.

A subsystem may also include other subsystems. A formal object-oriented formalism for specifying GRCs, and systems composed from reactive objects has been described in [AAM98].

2.1.2 Three-Tiered Formalism

The TROM formalism has three levels. Two of them correspond to the traditional OO approach to system development: to define classes, and then to instantiate them to compose subsystems. The TROM methodology adds one more level for defining data models. In the following sections we review the TROM formalism.

Figure 2 shows three-tiered structure of the TROM methodology introduced by Achuthan [Ach95]. The formalism is sufficiently expressive for modeling real-time reactive systems.

The three tiers independently specify system configurations, reactive objects, and abstract data types, by importing lower-tier specifications into upper tiers. Large and complex systems can be developed incrementally by composing, verifying, and integrating subsystems.

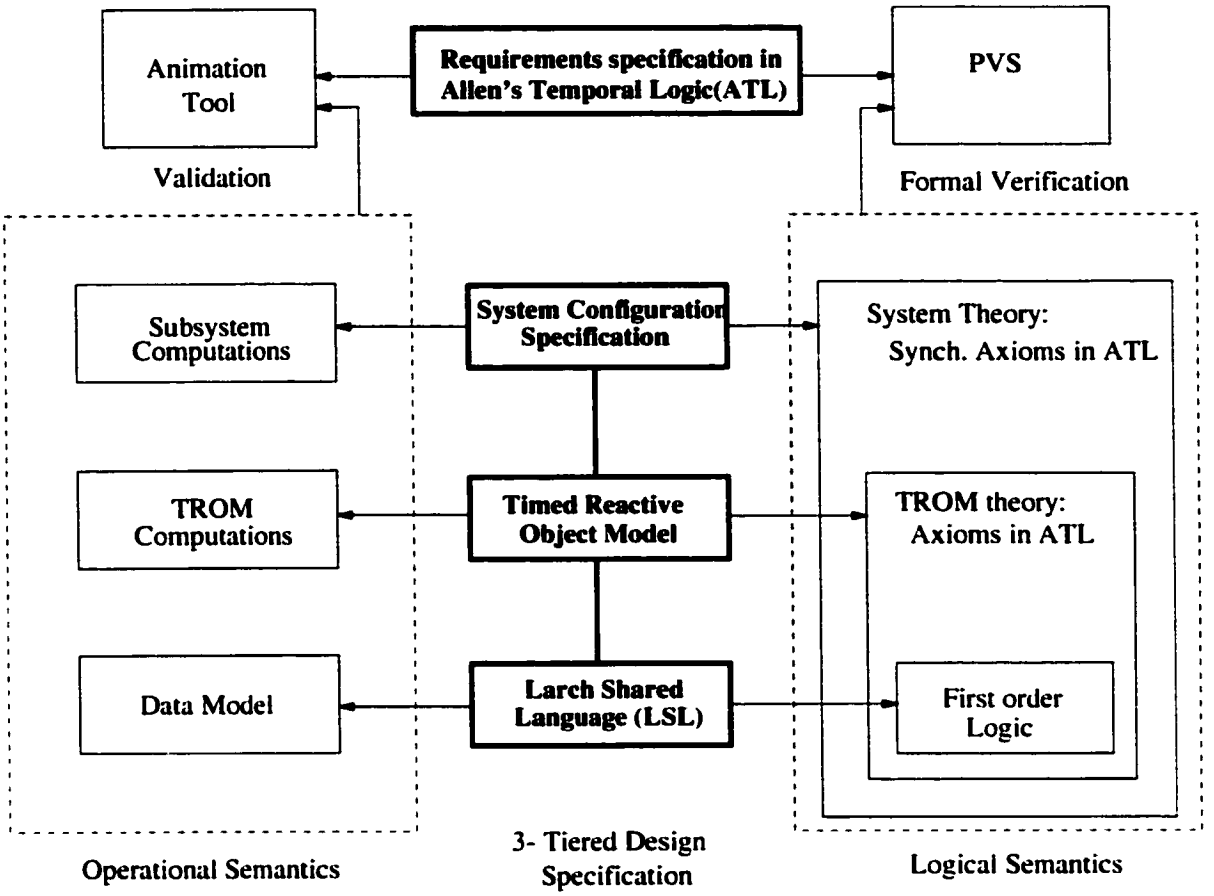


Figure 2: Overview of TROM methodology

First Tier - Larch Formalism

The Larch family of languages supports a *two-tiered*, definitional style of specification. Each specification has components written in two languages: one language that is designed for a specific programming language. The other language is independent of any programming language. The former kind is *Larch Interface Language* (LIL), and the latter is the *Larch Shared Language* (LSL).

LSL specifications define two kinds of symbols: *operators* and *sorts*. The concepts of operators and sorts are similar to the programming language concepts of procedure and type. Operators stand for total functions from tuples of values to values. Sorts stand for disjoint non-empty sets of values, and are used to indicate the domains and ranges of operators.

The *trait* is the basic unit of specification in LSL and abstractly specifies data in a definitional style. A trait introduces operators, defines the operators with a set of

equations that defines which terms are equal to one another, and may assert additional properties about the sorts and operators. A trait can also include other traits. Figure 3 shows the LSL trait defining the terms to represent values of the data model *Set*.

$Set(E, C)$: **trait**
  % Essential finite-set operators
  includes *Integer*
  introduces
    $\{\} :\rightarrow C$
    $insert : E, C \rightarrow C$
    $\_ \in \_ : E, C \rightarrow Bool$
    $delete : E, C \rightarrow C$
  asserts
    C **generated by** $\{\}, insert$
    C **partitioned by** $\in$
    $\forall\ s : C, e, e_1, e_2 : E$
      $\neg(e \in \{\})$
      $e_1 \in insert(e_2, s) == e_1 = e_2 \vee e_1 \in s$
      $delete(e_1, insert(e_2, s)) ==$ **if** $(e_1 = e_2)$ **then** s
        else $insert(e_2, delete(e_1, s))$
  implies
    $\forall\ e, e_1, e_2 : E, s : C$
      $insert(e, s) \neq \{\}$
      $insert(e, insert(e, s)) == insert(e, s)$
      $insert(e_1, insert(e_2, s)) ==$
        $insert(e_2, insert(e_1, s))$
    converts $\in, delete$
      exempting $\forall\ x : E$
      $delete(x, \{\})$

Figure 3: LSL Trait for Set

The *Set* trait introduces sorts E and C, begins by *includes* clause including another trait, Integer, which can be found in the LSL handbook [GH93]. The *introduces* clause declares a set of operators, each with its *signature* (the sorts of its domain and range). The *body* of the trait contains, following the *asserts* clause, equations between terms containing operations and variables. The *generated by* clause states that all values of sort C can be represented by terms $\{\}$ and *insert*. The operators listed in the *partitioned by* clause are sufficient to distinguish unequal set values. The *implies* section describes additional properties of sort C that follow from the assertions

part. The theory of a trait is the set of all logical consequences of its assertions. It contains everything that logically follows from its assertions, but nothing else. All operators listed in the *converts* clause are defined for terms in their domains in the exceptions noted in the *exempting* clause.

Second Tier - TROM Class

A TROM class is a hierarchical finite state machine augmented with ports, attributes, logical assertions on the attributes and time constraints, as shown in Figure 4. It is also called Generic Reactive Class (GRC). Such an object is assumed to have a single thread of control. A TROM object communicates with its environment by synchronous message passing, which occurs at a port.

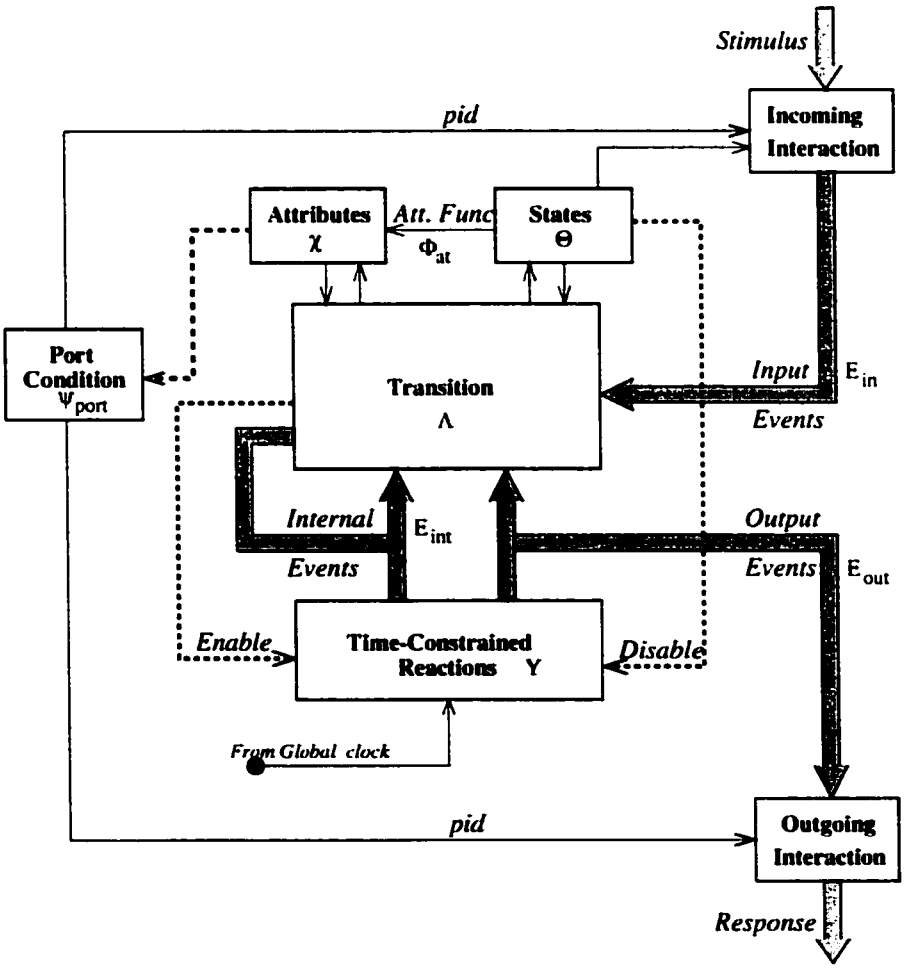


Figure 4: Anatomy of a reactive object

A TROM object consists of a set of events, states, attributes, attribute functions, transition functions and timing constraints. The events include internal, input and output events. For instance, e, e?, e! denote internal event, input event and external event respectively. Input and output events occur at a port and represent message passing.

The attributes of a TROM object are of two kinds: (i) abstract data types imported from the first tier, and (ii) port types. A port is an abstraction of an access point for a bi-directional communication link between a TROM and its environment. A TROM can have multiple port types associated with it. A port type denotes the set of messages (external events) that are allowed at a port of that particular type. The signature of a port-type P gives the set of events that can occur at the port-type P, denoted by $\mathcal{E}^P$. $\mathcal{E}^P = \mathcal{E}^P_{in} \bigcup \mathcal{E}^P_{out}$, where $\mathcal{E}^P_{in}$ is the set of input events, $\mathcal{E}^P_{out}$ is the set of output events, and $\mathcal{E}^P_{in} \bigcap \mathcal{E}^P_{out} = \emptyset$. Message exchange only occurs between compatible ports. Two port type P and Q are *compatible* if

- $e? \in \mathcal{E}^P_{in} \Leftrightarrow e! \in \mathcal{E}^Q_{out}$
- $e! \in \mathcal{E}^P_{out} \Leftrightarrow e? \in \mathcal{E}^Q_{in}$

The attribute functions define the association of attributes to states. For each state, the attribute function defines a subset of attributes that are active in the state. For a computation associated with a transition entering a state, only the attributes associated with that state are modified and all other attributes will be read-only in that computation.

Each specification describes the computational step associated with the occurrence of an event. A transition specification has three logical assertions: an enabling and a post-condition as in Hoare logic, and a port-condition specifying the port at which the transition can occur. The assertions may involve attributes and keyword *pid* for port identifier. A transition causes a *reaction* in the form of occurrence of either an internal event or an output event. There may be a timing constraint on the occurrence of the reaction.

A timing constraint can be associated with a transition to describe the time-constraint response to a stimulus.

A formal definition of the different components of a TROM object as described above is presented below.
A TROM object is an 8-tuple ($\mathcal{P}$, $\mathcal{E}$, Θ, $\mathcal{X}$, $\mathcal{L}$, Φ, Λ, Υ) such that:

- $\mathcal{P}$ is a finite set of port-types with a finite set of ports associated with each port-type. A distinguished port-type is the null-type $P_\circ$ whose only port is the null port $\circ$.

- $\mathcal{E}$ is a finite set of events and includes the silent-event tick. The set $\mathcal{E} - \{$ tick $\}$ is partitioned into three disjoint subsets: $\mathcal{E}_{in}$ is the set of input events, $\mathcal{E}_{out}$ is the set of output events, and $\mathcal{E}_{int}$ is the set of internal events. Each $e \in (\mathcal{E}_{in} \bigcup \mathcal{E}_{out})$, is associated with a unique port-type $P \in \mathcal{P} - \{\mathcal{P}_\circ\}$.

- Θ is a finite set of states. $\theta_0 \in \Theta$, is the *initial* state.

- $\mathcal{X}$ is a finite set of typed attributes. The attributes can be of one of the following two types: i) an abstract data type; ii) a port reference type.

- $\mathcal{L}$ is a finite set of LSL traits introducing the abstract data types used in $\mathcal{X}$.

- Φ is a function-vector (Φ_s, Φ_{at}) where,

  - $\Phi_s : \Theta \rightarrow 2^\Theta$ associates with each state θ a set of states, possibly empty, called *substates*. A state θ is called *atomic*, if $\Phi_s(\theta) = \emptyset$. By definition, the initial state θ_0 is atomic. For each non-atomic state θ, there exists a unique atomic state $\theta^* \in \Phi_s(\theta)$, called the entry-state.

  - $\Phi_{at} : \Theta \rightarrow 2^\mathcal{X}$ associates with each state θ a set of attributes, possibly empty, called the *active* attribute set. At each state θ, the set $\overline{\Phi_{at}}(\theta) = \mathcal{X} - \Phi_{at}(\theta)$ is called the *dormant* attribute set of θ.

- Λ is a finite set of *transition specifications* including λ_{init}. A transition specification $\lambda \in \Lambda - \{\lambda_{init}\}$, is a three-tuple : $< \langle \theta, \theta' \rangle$; $e(\varphi_{port})$; $\varphi_{en} \Longrightarrow \varphi_{post} >$; where:

  - $\theta, \theta' \in \Theta$ are the source and destination states of the transition;

  - event $e \in \mathcal{E}$ labels the transition; φ_{port} is an assertion on the attributes in $\mathcal{X}$ and a reserved variable *pid*, which signifies the identifier of the port

at which an interaction associated with the transition can occur. If $e \in \varepsilon_{int} \bigcup \{\mathsf{tick}\}$, then the assertion φ_{port} is absent and e is assumed to occur at the null-port $\circ$.

- φ_{en} is the enabling condition and φ_{post} is the postcondition of the transition. φ_{en} is an assertion on the attributes in $\mathcal{X}$ specifying the condition under which the transition is enabled. φ_{post} is an assertion on the attributes in $\mathcal{X}$, primed attributes in $\Phi_{at}(\theta')$ and the variable *pid*, and it implicitly specifies the data computation associated with the transition.

For each $\theta \in \Theta$, the silent-transition $\lambda_{s\theta} \in \Lambda$ is such that,

$$\lambda_{s\theta} : \langle \theta, \theta \rangle; \mathsf{tick}; true \Longrightarrow \forall x \in \Phi_{at}(\theta) : x = x';$$

The initial-transition λ_{init} is such that $\lambda_{init} : \langle \theta_0 \rangle;\ Create();\ \varphi_{init}$ where φ_{init} is an assertion on active-attributes of θ_0.

- Υ is a finite set of *time-constraints*. A timing constraint $v_i \in \Upsilon$ is a tuple $(\lambda_i, e_i', [l, u], \Theta_i)$ where,
  - $\lambda_i \neq \lambda_s$ is a transition specification.
  - $e_i' \in (\mathcal{E}_{out} \bigcup \mathcal{E}_{int})$ is the *constrained event*.
  - $[l, u]$ defines the minimum and maximum response times.
  - $\Theta_i \subseteq \Theta$ is the set of disabling states when the object enters one of these states, timing constraint v_i will be ignored.

Figure 5 shows the template for a class specification.
Figure 6 is an example of the formal specification of the TROM class *Arbiter*. An arbiter allocates shared resources to processes requesting them. It enqueues the requests for a resource received from processes and allocates the resource to the next process waiting in the queue.

Third Tier - System Configuration Specification

Each subsystem is the collaboration of the objects instantiated from the second tier. A System Configuration Specification (SCS) is defined to specify a reactive system or

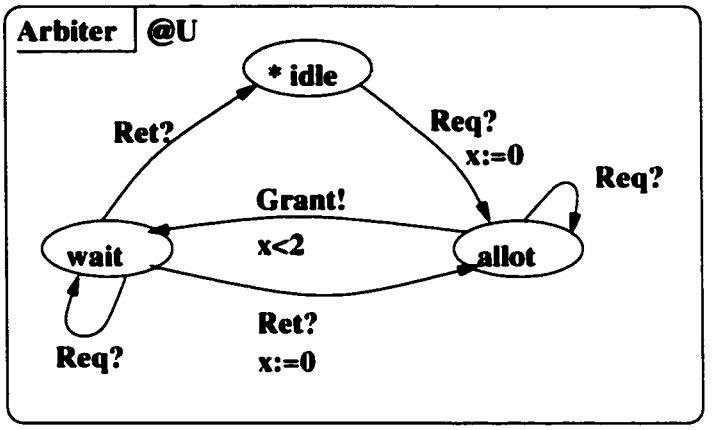
```
Class < name >
    Events:
    States:
    Attributes:
    Traits:
    Attribute-Function:
    Transition-Specifications:
    Time-Constraints:
end
```

Figure 5: Template for Class Specification

subsystem by composing reactive objects or by composing smaller subsystems. Figure 7 shows the template for a system configuration specification.

2.1.3 Operational Semantics

Reactive objects in a system communicate through messages. A message from an object to another object in the system is called a *signal* and is represented by a tuple $\langle e, p_i, t \rangle$, denoting that the event e occurs at time t, at a port p_i. The *status* of a TROM at any time t, is the tuple $(\theta; \vec{a}; \mathcal{R})$, where the current state θ is a simple state of the TROM , $\vec{a}$ is the assignment vector, and $\mathcal{R}$ is the vector of outstanding reactions. A *computational step* of a TROM occurs when the object with status $(\theta; \vec{a}; \mathcal{R})$, receives a signal $\langle e, p_i, t \rangle$ and there exists a transition specification that can change the status of the TROM . A computation c of a TROM object $\mathcal{A}$ is a sequence, possibly infinite, of alternating statuses and signals, $\mathcal{OS}_0 \overset{\langle e_0, p_0, t_0 \rangle}{\rightarrow} \mathcal{OS}_1 \overset{\langle e_1, p_1, t_1 \rangle}{\rightarrow} \ldots$. A reactive system may not terminate; consequently, a computation is in general an infinite sequence. The set of all computations of a TROM object $\mathcal{A}$ is denoted by by $Comp(\mathcal{A})$. The computation of a subsystem is an infinite sequence of system statuses and signals that effect status changes [AAM98]. A *period* is a finite subsequence of the computation such that it starts with the initial state and finishes with its next appearance in the sequence.



Class *Arbiter* [@U]
Events: *Req?U, Grant!U, Ret?U*
States: **idle, allot, wait*
Attributes: *rqQueue: UQueue; hold:@U*
Traits: *Queue[@U, UQueue]*
Attribute-function:
allot $\mapsto$ *rqQueue; wait* $\mapsto$ *rqQueue, hold*;
Transition-Specifications:
R_1 : $\langle$*idle, allot*$\rangle$; *Req?(true)*;
true $\Longrightarrow$ *rqQueue'* = *insert*(pid, {});
R_2 : $\langle$*allot, wait*$\rangle$; *Grant!*(pid $\in$ *rqQueue*);
true $\Longrightarrow$ *rqQueue' = tail(reQueue))* $\wedge$ *(hold'* = pid);
R_3 : $\langle$*allot, allot*$\rangle$, $\langle$*wait, wait*$\rangle$; *Req?(not* pid $\in$ *rqQueue)*;
true $\Longrightarrow$ *rqQueue' = insert*(pid, *rqQueue*);
R_4 : $\langle$*wait, allot*$\rangle$; *Ret?* (pid *hold*);
$\neg$ *isEmpty(rqQueue)* $\Longrightarrow$ *equal(rqQueue', rqQueue)*;
R_5 : $\langle$*wait, idle*$\rangle$; *Ret?* (pid = *hold*);
isEmpty(rqQueue) $\Longrightarrow$ *true*;
Time-Constraints:
TC_1 : (R_1, *Grant*, [0,2], $\emptyset$)
TC_2 : (R_4, *Grant*, [0,2], $\emptyset$)
end

Figure 6: Formal Specification of Class Arbiter

```
Subsystem < name >
    Include:
    Instantiate:
    Configure:
end
```

Figure 7: Template for System Configuration Specification

2.1.4 Current TROMLAB Architecture

TROMLAB is a framework based on TROM formalism for object-oriented design and development of real-time reactive systems. Figure 8 is an overall architectural view of TROMLAB.

The current TROMLAB development environment includes the following components:

- **Rose-GRC Translator -** [Pop99] which automatically maps the graphic Rose model to the formal specification;
- **Interpreter -** [Tao96] which parses, syntactically checks a specification and constructs an internal representation;
- **Simulator -** [Mut96] which animates a subsystem based on the internal representation, and enables a systematic validation of the specified system;
- **Browser for Reuse -** [Nag99] which is an interface to a library, to help users navigate, query and access various system components for reuse during system development;
- **Graphical User Interface -** [Sri99] which is a visual modeling and interaction facility for a developer using the TROMLAB environment;
- **Reasoning System -** [Hai99] which provides a means of debugging the system during animation by facilitating interactive queries of hypothetical nature on system behavior.
- **Verification Assistant -** [Pom99] which is an automated tool that enables mechanized axiom extraction from real-time reactive systems.

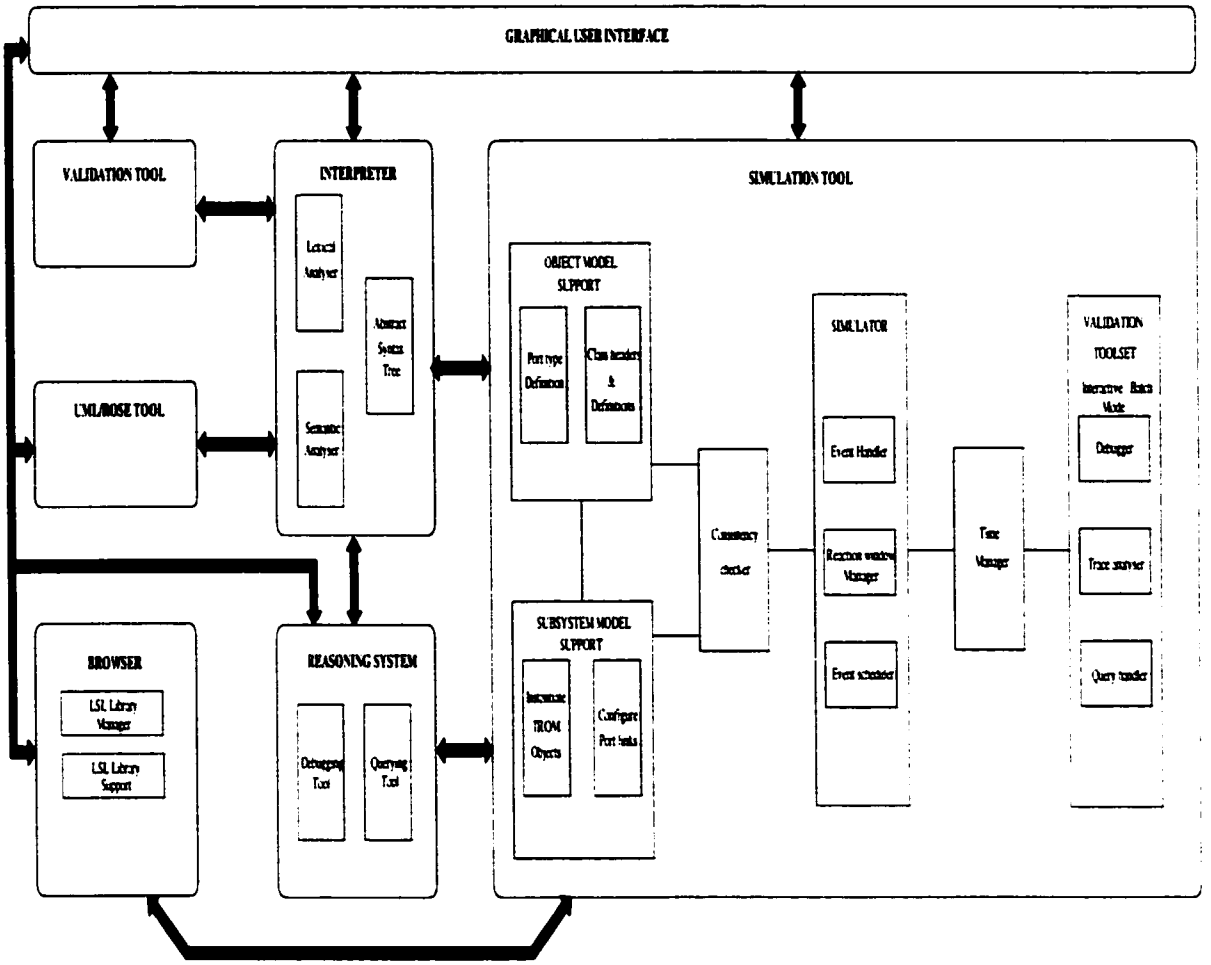


Figure 8: Existing TROMLAB architecture

The TROM formalism is illustrated on the railroad crossing problem has been considered as a bench mark example by researchers in real-time systems community.

2.2 Train-Gate-Controller Case Study

According to the problem description, several trains cross a gate independently and simultaneously using non-overlapping tracks. A train chooses the gate it intends to cross; there is a unique controller monitoring the operations of each gate. When the first train approaches a gate, it sends a message to the corresponding controller, which then commands the gate to close. When the last train crossing a gate leaves the crossing, the controller commands the gate to open. The safe operation of the controller depends on the satisfaction of certain timing constraints, so that the gate

is closed before a train enters the crossing, and the gate is opened after the last train leaves the crossing.

Assumptions

- Trains inform the controller before entering the crossing
- Trains inform the controller when leaving the crossing
- Assumption of perfect technology

Specific Requirements

In the specification of the reactive systems, two important behavioral properties needs to be formally expressed: *safety* and *liveness*. Informally, a safety property implies that *something bad will not happen*, and a liveness property implies that *something good will eventually happen*.

Safety Requirements

Whenever there is a train inside a crossing, the gate remains closed.

Liveness Requirements

When the last train leaves the crossing, the gate eventually reopens.

Timing Requirements

Timing constraints include the maximum and the minimum times required for a train to be in the gate from the instant it was observed by a controller, the maximum permitted time for a gate to open or close, and the time bounds for a train to cross a gate.

> A train enters the crossing within an interval of 2 to 4 time units after having indicated its presence to the controller.
>
> The train informs the controller that it is leaving the crossing within 6 time units of sending the approaching message.

The controller instructs the gate to close within 1 time unit of receiving an approaching message from the first train entering the crossing, and starts monitoring the gate.

The controller continues to monitor the closed gate if it receives an approaching message from another train.

The controller instructs the gate to open within 1 time unit of receiving a message from the last train to leave the crossing.

The gate must close within 1 time unit of receiving instructions from the controller.

The gate must open within an interval of 1 to 2 time units of receiving instructions from the controller.

Railroad Crossing System – Problem Analysis

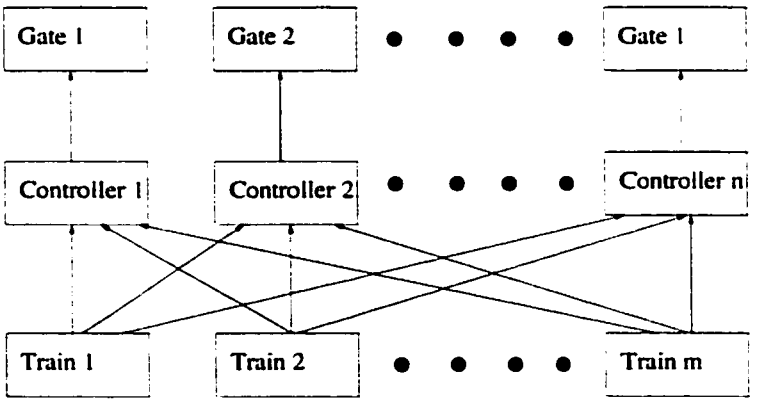


Figure 9: Railroad Crossing System - Problem Analysis

In our model we allow many trains to communicate with one controller, and vice versa and there is one controller for each gate (Figure 9).

Objects

The objects are abstracting the real-world entities that belong to the environment, to the system, or are communication channels between the environmental and the system objects.

- **Environmental objects**

- *trains*
- *gates*

- **System objects**
  - *controllers*

- **Communication channels:**
  - trains to controllers through *sensors* - **many-to-many**
  - controllers to gates through *actuators* - **one-to-one.**

Formal Behavioral Specifications

The formal behavioral specifications are based on the TROM formalism and the Real-Time UML (RTUML).
The LSL traits model the data structures.
The RTUML is used to model the objects and the system in order to describe the system boundary:

- The use-case model (use-case diagram(s) and sequence diagram(s)) helps to understand the external system behavior. The sequence diagram(s) specify the dynamic behavior in conformance with the timing constraints.
- The GRC classes aggregate instances of data models. Each GRC has a statechart diagram.
- The statechart diagrams addresses the logical assertions on transitions and timing constraints on reactions. Each statechart diagram is annotated with a class specification, a formal description of the class diagram. and operational semantics expressed in first-order logic.
- The main class diagram describe the classes and their relationships.
- The RTUML's *operational semantics* describe formally the object and system behavior, and form the foundation for the formal validation and verification of safety and liveness properties of the system.

- The collaboration diagram models the system configuration composed from instances of object models, and channels for object communication.

We specify the finite state machine behavior of the train, gate, controller objects using temporal relations expressed in predicate logic.
Each requirement is stated as a first-order logical formula with explicit *time* over predicates modeling events and actions. At this level of abstraction, an event corresponds to a high-level action, as opposed to a transition event.

Interpretations for predicates and variables are given over the domain considered in the application.
The predicates, variables and their interpretations are as follows:

- $was\_at(s, t)$ - the object was in state s at time t.
- $occur(e, t)$ - the event e occurred at time t.

The timing constraints formulas for the three classes are given in the corresponding object model sections.
The synchronization timing constraints for each port link defined in a SCS (subsystem configuration specification), are given in the system configuration section.
The predicate formulas formalizing the safety and liveness properties are given in the subsystem section.

Main Class Diagram

The UML model for the Train-Gate-Controller problem has three generic reactive classes: *Train*, *Controller*, and *Gate*. These three classes and their relationships are described in one main class diagram, as depicted in Figure 10.
Train GRC is an aggregate of port types *@C*.
Controller GRC is an aggregate of port types *@G* and *@P*.
Gate GRC is an aggregate of port types *@S*.
There is an association between port type *@C* of *Train* and *@P* of *Controller*, meaning that the generic class *Train* uses port type *@C* to communicate to the generic class *Controller* through its port type *@P*.

There is also an association between port type *@S* of *Gate* and port type *@G* of *Controller*, meaning that generic class *Controller* uses port type *@G* to communicate with generic class *Gate* through its port type *@S*.
Train GRC has one port type *@C*. At port type *@C*, the following events may occur: output event *Near*, output event *Exit*.

Train GRC has one attribute, named *cr*, whose type is the port type *@C*.
Controller GRC has two port types: *@P* and *@G*. At port type *@P*, the following events may occur: input event *Near*, input event *Exit*. At port type *@G*, the following events may occur: output event *Lower*, output event *Raise*.
Controller GRC has one data type attribute, *inSet*. The type is an abstract data type defined in the LSL trait *Set* with parameters *@P* and *PSet*, where *@P* is the type of each name and *PSet* is the name of the abstract data type.
Gate GRC has one port type*@S*. At port type *@S* the following events may occur: input event *Lower*, input event *Raise*.

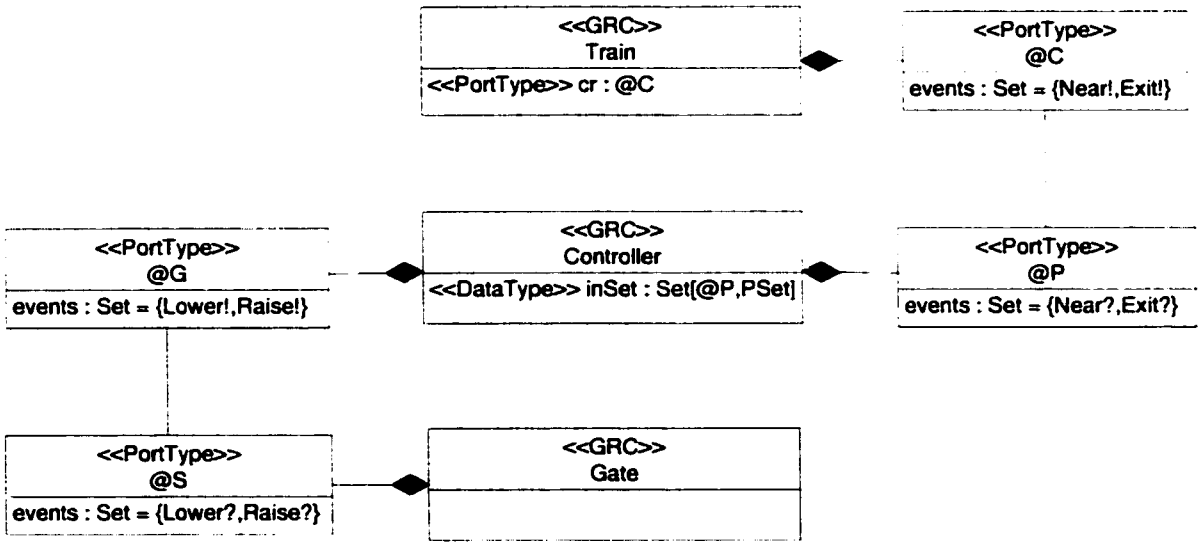


Figure 10: Main Class Diagram for Train-Gate-Controller

Real-time features such as minimal and maximal delays, exact event occurrences, are been specified in this framework.

Train Model

The statechart diagram for *Train* in shown in Figure 11.

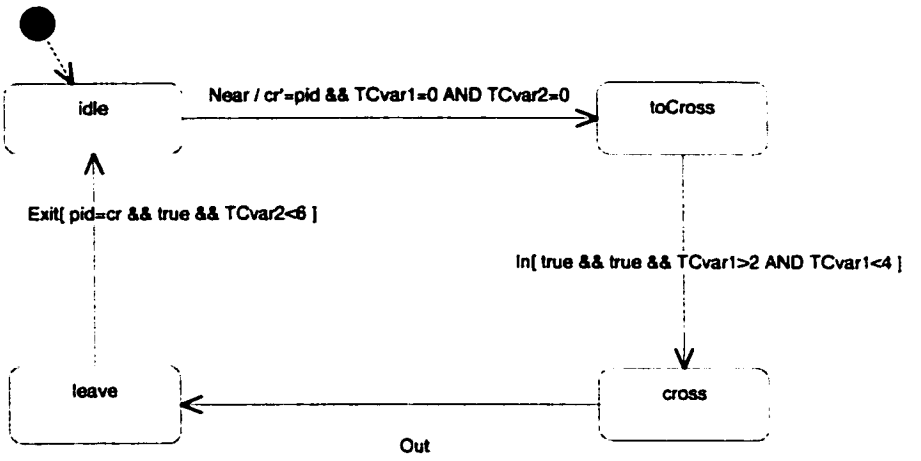


Figure 11: Statechart Diagram for Train

A *Train* object can be in one of four states: *idle*, *toCross*, *cross*, *leave*. *Idle* is the initial state.

When event *Near* occurs in state *idle*, attribute *cr* is set to *pid*, the identifier of the port where *Near* occurs. This transition is the constraining transition for two time constraints, labeled *TCvar1* and *TCvar2*. *Train* goes into state *toCross*.

A transition from state *toCross* to *cross* happens when internal event *In* occurs in state *toCross*, and if the time constraint condition $TCvar1 \geq 2$ AND $TCvar1 \leq 4$ is true. This time constraint means that internal event *In* should occur within 2 to 4 time units after event *Near* occurs in state *idle*.

When internal event *Out* occurs in state *cross*, *Train* goes into state *leave*.

A transition from state *leave* to *idle* happens when event *Exit* occurs in state *leave*, if the attribute *cr* has the value *pid* (pid is the identifier of the port where *Exit* occurs), and if the time constraint condition $Tcvar2 \leq 6$ is true. This time constraint means that event *Exit* should occur within 6 time units after event *Near* occurs in state *idle*.

Train: Formal Description The formal description for *Train* in shown in Figure 12.

Train: Operational Semantics

- S_1: train is in state *Idle*
- S_2: train is in state *toCross*

```
Class Train [@C]
Events: Near!@C, Out, Exit!@C, In
States: *idle, cross, leave, toCross
Attributes: cr:@C
Traits:
Attribute-Function: idle -> {};cross -> {};leave -> {};toCross -> {cr};
Transition-Specifications:
   R1: <idle,toCross>; Near(true); true => cr'=pid;
   R2: <cross,leave>; Out(true); true => true;
   R3: <leave,idle>; Exit(pid=cr); true => true;
   R4: <toCross,cross>; In(true); true => true;
Time-Constraints:
   TCvar2: R1, Exit, [0, 6], {};
   TCvar1: R1, In, [2, 4], {};
end
```

Figure 12: Formal Description of Class Train

- S_3: train is in state *cross*
- S_4: train is in state *leave*
- Time constraints for the class Train:

1. $was\_at(S_2, t_1) \wedge \forall t \bullet ((t_1 < t < t_2) \wedge \neg was\_at(S_3, t)) \wedge was\_at(S_3, t_2) \wedge t_2 - t_1 > 2 \wedge t_2 - t_1 < 4$
2. $was\_at(S_2, t_1) \wedge \forall t \bullet ((t_1 < t < t_2) \wedge \neg was\_at(S_1, t)) \wedge was\_at(S_1, t_2) \wedge t_2 - t_1 > 0 \wedge t_2 - t_1 < 6$

Controller Model

The statechart diagram for *Controller* is shown in Figure 13.
A *Controller* object can be in one of four states: *idle*, *activate*, *monitor*,*deactivate*. *Idle* is the initial state.

When event *Near* occurs in state *idle*, the attribute *inSet* is modified to include the new entry *pid* (*pid* is the identifier of the port where *Near* occurs). The *Controller* goes into state *activate*. This transition is the constraining transition for time constraint *TCvar1*.
When event *Near* occurs in state *activate* from a different *Train* (*pid* is not already a member of set *inSet*), the attribute *inSet* is modified to include the new *pid* (identifier

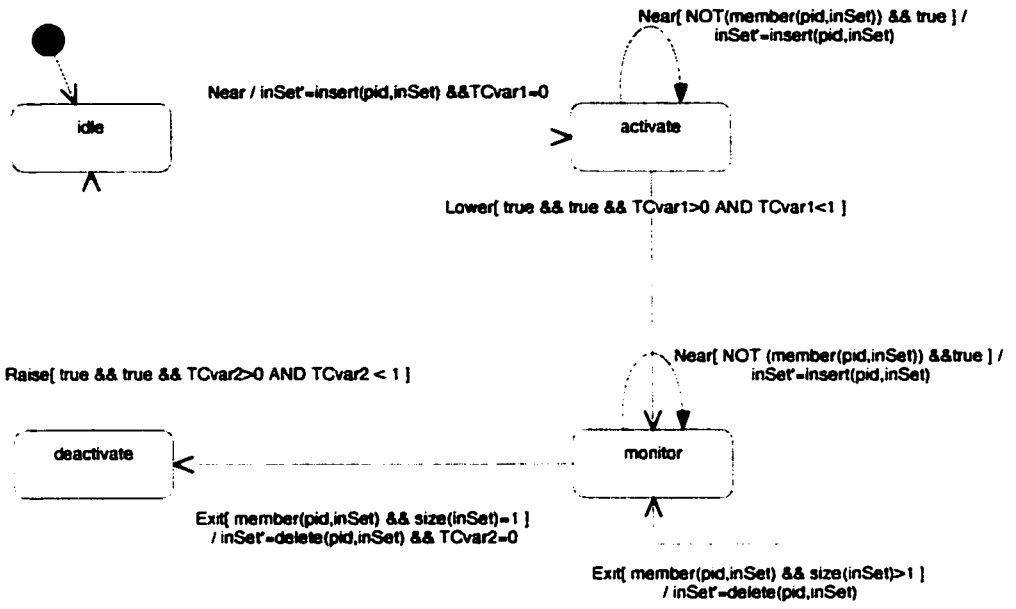


Figure 13: Statechart Diagram for Controller

of the port where the new event *Near* occurs). The *controller* remains in the state *activate.*

When event *Lower* occurs in state *activate*, if the time constraint condition $TCvar1 \leq 1$ is true, the *Controller* goes into state *monitor.* This time constraint means that event *Lower* should occur within one time unit after event *Near* occurs in state *idle.*

When event *Near* occurs in state *monitor* from a different *Train* (*pid* is not already a member of set *inSet*), the attribute *inSet* is modified to include the new *pid* (identifier of the port where the new event *Near* occurs). The *Controller* remains in state *monitor.*

When event *Exit* occurs in state *monitor*, if the identifier (*pid*) of the port where event *Exit* was received is a member of *inSet* and if the size of *inSet* is greater than 1, meaning that more than one *Trains* are in the crossing, then the current *pid* is deleted from *inSet* and *Controller* remains in state *monitor.*

When event *Exit* occurs in state *monitor*, if the identifier (*pid*) of the port where event *Exit* was received is a member of *inSet* and if the size of *inSet* is equal to 1, meaning that this is the only *Train* in the crossing, then the current *pid* is deleted from *inSet* and *Controller* goes into state *deactivate.* This is the constraining transition for time constraint *TCvar2.*

When event *Raise* occurs in state *deactivate*, if time constraint condition $Tcvar2 \leq 1$

is true, the *Controller* goes into state *idle*. This time constraint condition means that event *Raise* should occur within 1 time unit after event *Exit* was received from the last *Train* in the crossing.

Controller: Formal Description The formal description for *Controller* in shown in Figure 14.

```
Class Controller [@P, @G]
Events: Lower!@G, Near?@P, Raise!@G, Exit?@P
States: *idle, activate, deactivate, monitor
Attributes: inSet:PSet
Traits: Set[@P,PSet]
Attribute-Function: activate -> {inSet};deactivate -> {inSet};monitor -> {inSet};idle -> {};
Transition-Specifications:
   R1: <activate,monitor>; Lower(true); true => true;
   R2: <activate,activate>; Near(!(member(pid,inSet))); true => inSet'=insert(pid,inSet);
   R3: <deactivate,idle>; Raise(true); true => true;
   R4: <monitor,deactivate>; Exit(member(pid,inSet)); size(inSet)=1 => inSet'=delete(pid,inSet);
   R5: <monitor,monitor>; Exit(member(pid,inSet)); size(inSet)>1 => inSet'=delete(pid,inSet);
   R6: <monitor,monitor>; Near(!(member(pid,inSet))); true => inSet'=insert(pid,inSet);
   R7: <idle,activate>; Near(true); true => inSet'=insert(pid,inSet);
Time-Constraints:
   TCvar1: R7, Lower, [0, 1], {};
   TCvar2: R4, Raise, [0, 1], {};
end
```

Figure 14: Formal Specification for Controller Class

Controller GRC: Operational Semantics

- C_1: controller is in state *idle*
- C_2: controller is in state *activate*
- C_3: controller is in state *monitor*
- C_4: controller is in state *deactivate*

1. $was\_at(C_2, t_1) \wedge \forall t \bullet ((t_1 < t < t_2) \wedge \neg was\_at(C_3, t)) \wedge was\_at(C_3, t_2) \wedge t_2 - t_1 > 0 \wedge t_2 - t_1 < 1$
2. $was\_at(C_4, t_1) \wedge \forall t \bullet ((t_1 < t < t_2) \wedge \neg was\_at(C_1, t)) \wedge was\_at(C_1, t_2) \wedge t_2 - t_1 > 0 \wedge t_2 - t_1 < 1$

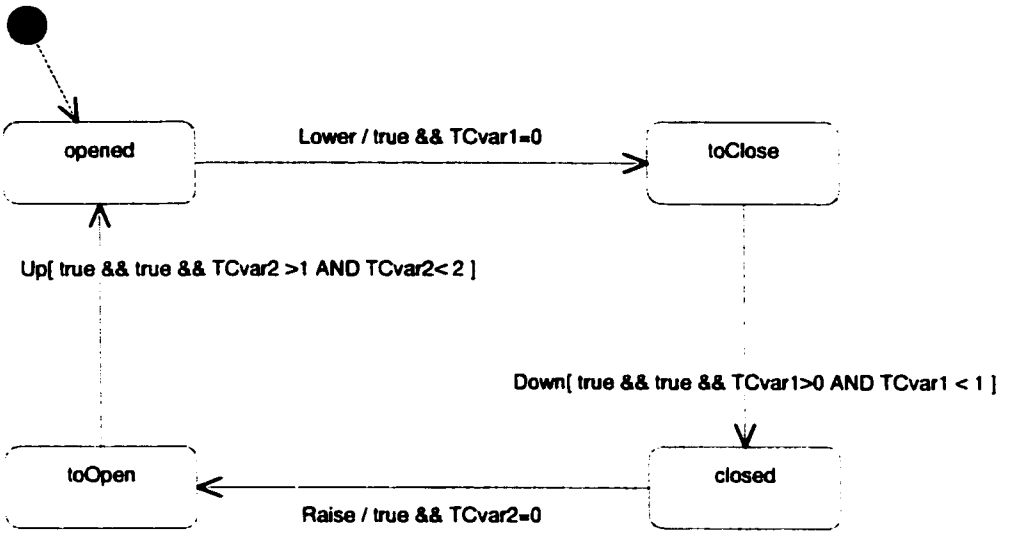


Figure 15: Statechart Diagram for Gate

Gate Model

The statechart diagram for *Gate* is shown in Figure 15.

A *Gate* object can be in one of four states: *opened*, *toClose*, *closed*, *toOpen*. *Closed* is the initial state.

When event *Lower* occurs in state *opened*, the *Gate* goes into state *toClose*. This is the constraining transition for time constraint labeled *Tcvar1*.

A transition from state *toClose* to *closed* happens when internal event *Down* occurs in state *toClose* if the time constraint condition $TCvar1 \leq 1$ is true. This time constraint means that internal event *Down* should occur within 1 time unit after event *Lower* occurs in state *opened*.

When event *Raise* occurs in state *closed*, the *Gate* goes into state *toOpen*. This is the constraining transition for time constraint *TCvar2*.

A transition from state *toOpen* to *open* happens when internal event *Up* occurs in state *toOpen* if the time constraint condition $TCvar2 \geq 1$ and $Tcvar2 \leq 2$ is true. This time constraint means that internal event *Up* should occur within 1 to 2 time units after event *Raise* occurs in state *closed*.

Gate: Formal Description The formal description for *Gate* in shown in Figure 16.

```
Class Gate [@S]
Events: Lower?@S, Down, Up, Raise?@S
States: *opened, toClose, toOpen, closed
Attributes:
Traits:
Attribute-Function: opened -> {};toClose -> {};toOpen -> {};closed -> {};
Transition-Specifications:
   R1: <opened,toClose>; Lower(true); true => true;
   R2: <toClose,closed>; Down(true); true => true;
   R3: <toOpen,opened>; Up(true); true => true;
   R4: <closed,toOpen>; Raise(true); true => true;
Time-Constraints:
   TCvar1: R1, Down, [0, 1], {};
   TCvar2: R4, Up, [1, 2], {};
end
```

Figure 16: Formal Description for Gate Class

Gate: Operational Semantics

- G_1: controller is in state *opened*
- G_2: controller is in state *toClose*
- G_3: controller is in state *closed*
- G_4: controller is in state *toOpen*

1. $was\_at(G_2, t_1) \wedge \forall t \bullet ((t_1 < t < t_2) \wedge \neg was\_at(G_3, t)) \wedge was\_at(G_3, t_2) \wedge t_2 - t_1 > 0 \wedge t_2 - t_1 < 1$
2. $was\_at(G_4, t_1) \wedge \forall t \bullet ((t_1 < t < t_2) \wedge \neg was\_at(G_1, t)) \wedge was\_at(G_1, t_2) \wedge t_2 - t_1 > 1 \wedge t_2 - t_1 < 2$

System Model

The (sub)systems has a Collaboration Diagram, a formal description, and synchronization semantics expressed in first-order logic. The safety and liveness properties are expressed in first-order logic. They have to be consistent with the timing and synchrony constraints.

A collaboration diagram depicts a system, or, for larger systems, a subsystem. We specify a system with one train, one controller and one gates, named *Train-Gate-Controller1* (Figure 17).

We also specify a system with 5 trains, 2 controllers and two gates, named *Train-Gate-Controller2* Figure 18. In this configuration, the object *train3* is allowed to interact with both controllers, while the other train objects can only interact with one of the controllers. This schematic drawing conforms to a system with different trains on specific routes. The collaboration diagram for this subsystem is shown in Figure 18.

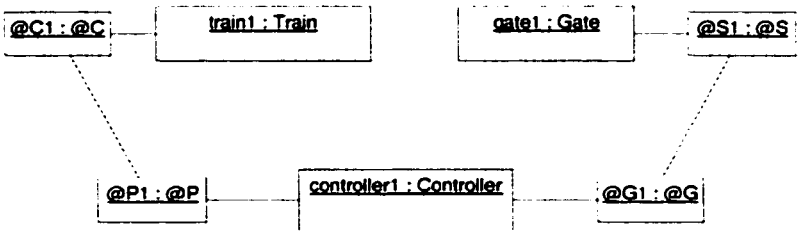


Figure 17: Collaboration Diagram for a Train-Gate-Controller1 Subsystem.

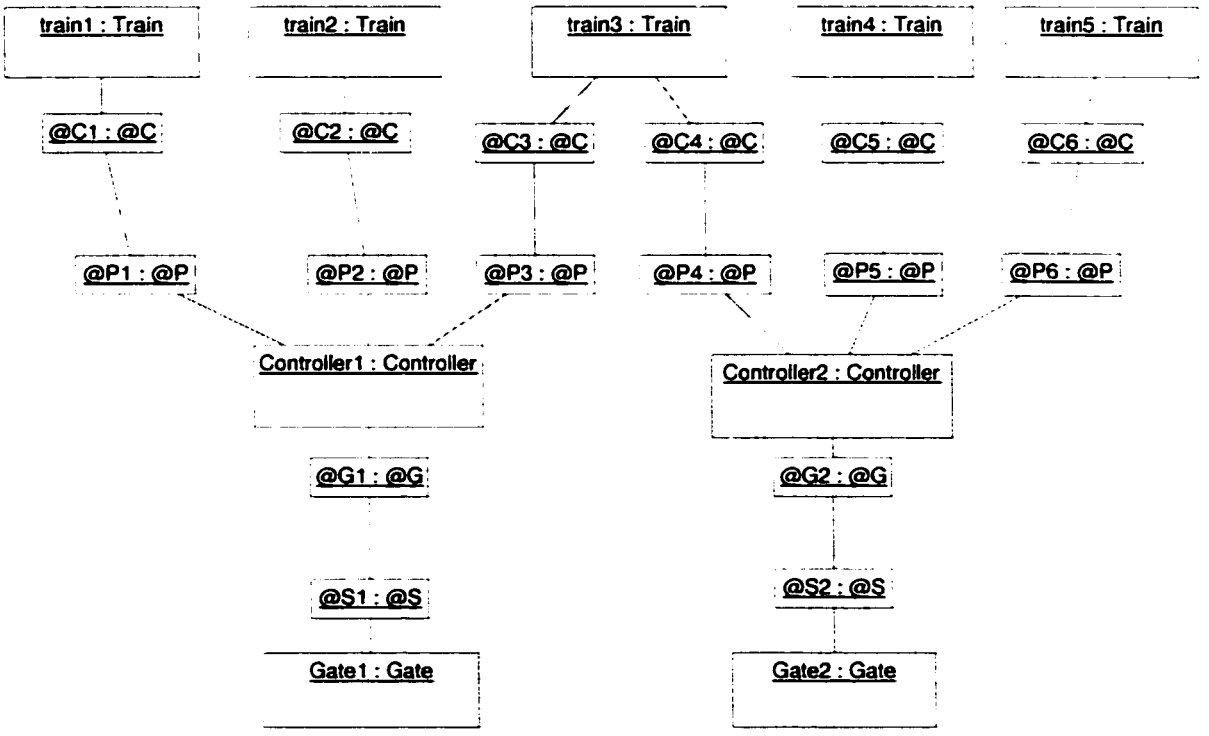


Figure 18: Collaboration Diagram for a Train-Gate-Controller2 Subsystem.

Sequence Diagrams

A sequence diagram depicts one possible scenario for the system (Figure 19). We specify a scenario of the system with one train, one controller and one gate, named *Train-Gate-Controller1* (Figure 17).

When a train approaches a gate, it sends a message to the corresponding controller, which then commands the gate to close. When the train crossing a gate leaves the crossing, a train sends a message to the controller, and the controller commands the gate to open.

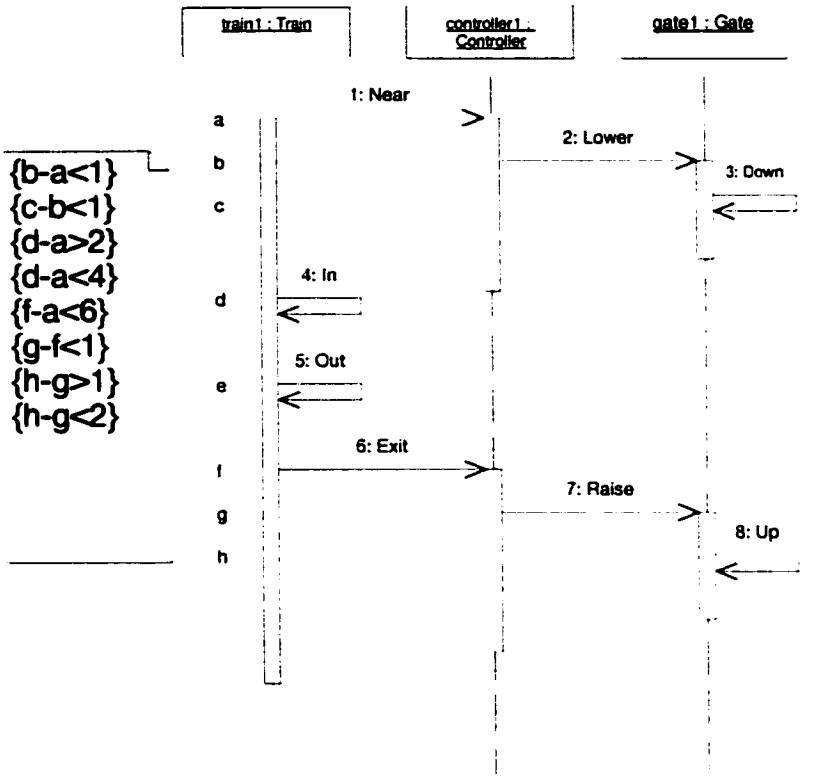


Figure 19: Sequence Diagram

Subsystem Formal Description

The formal specification for the subsystem configuration described in Figure 17 (1 Train object, 1 Controller object, and 1 Gate object) is as shown in Figure 20.

The formal specification for the subsystem configuration described in Figure 18 (5 Train objects, 2 Controller objects, and 2 Gate objects) is as shown in Figure 21.

```
SCS TrainGateController
  Includes:
  Instantiate:
    gate1::Gate[@S:1];
    train1::Train[@C:1];
    controller1::Controller[@P:3, @G:1];
  Configure:
    controller1.@G1:@G <-> gate1.@S1:@S;
    controller1.@P1:@P <-> train1.@C1:@C;
end
```

Figure 20: Formal specification for Train-Gate-Controller1 Subsystem.

Synchronization Semantics for Train-Gate-Controller1 Subsystem

- *Synchronization between Train and Controller*

  1. $was\_at(S_1, t) \wedge was\_at(C_1, t) \wedge occur(Near, t) \Rightarrow was\_at(S_2, t') \wedge was\_at(C_2, t') \wedge (t' = t + \epsilon)$
  2. $was\_at(S_1, t) \wedge was\_at(C_2, t) \wedge occur(Near, t) \Rightarrow was\_at(S_2, t') \wedge was\_at(C_2, t') \wedge (t' = t + \epsilon)$
  3. $was\_at(C_3, t) \wedge was\_at(S_1, t) \wedge occur(Near, t) \Rightarrow was\_at(C_3, t') \wedge was\_at(S_2, t') \wedge (t' = t + \epsilon)$
  4. $was\_at(C_3, t) \wedge was\_at(S_4, t) \wedge occur(Exit, t) \Rightarrow (TCvar2 > 0 \wedge was\_at(C_3, t') \wedge was\_at(S_1, t') \wedge (t' = t + \epsilon)) \vee (TCvar2 = 0 \wedge was\_at(C_4, t') \wedge was\_at(S_1, t') \wedge (t' = t + \epsilon))$

- *Synchronization between Controller and Gate*

  1. $was\_at(C_2, t) \wedge was\_at(G_1, t) \wedge occur(Lower, t) \Rightarrow was\_at(C_3, t) \wedge was\_at(G_2, t) \wedge (t' = t + \epsilon)$
  2. $was\_at(C_4, t) \wedge was\_at(G_3, t) \wedge occur(Raise, t) \Rightarrow was\_at(C_1, t') \wedge was\_at(G_4, t') \wedge (t' = t + \epsilon)$

Safety and Liveness Properties

Consider an interval of time $[t_1, t_2]$, such that at time t_1 there is no train in the crossing and the first train enters the crossing, and at time t_2 a train leaves the crossing and

```
SCS TrainGateController
  Includes:
  Instantiate:
    gate1::Gate[@S:1];
    gate2::Gate[@S:1];
    train1::Train[@C:1];
    train2::Train[@C:1];
    train3::Train[@C:2];
    train4::Train[@C:1];
    train5::Train[@C:1];
    controller1::Controller[@P:3, @G:1];
    controller2::Controller[@P:3, @G:1];
  Configure:
    controller1.@G1:@G <-> gate1.@S1:@S;
    controller2.@G2:@G <-> gate2.@S2:@S;
    controller1.@P1:@P <-> train1.@C1:@C;
    controller1.@P2:@P <-> train2.@C2:@C;
    controller1.@P3:@P <-> train3.@C3:@C;
    controller2.@P4:@P <-> train3.@C4:@C;
    controller2.@P5:@P <-> train4.@C5:@C;
    controller2.@P6:@P <-> train5.@C6:@C;
end
```

Figure 21: Formal specification for Train-Gate-Controller2 Subsystem.

there is no other train still in the crossing.

Safety Property: For a safe functional and temporal behavior of the system, the gate must be closed whenever there is a train inside the crossing.
Formally,
$was\_at(S_2, t_1) \wedge was\_at(S_4, t_2) \wedge (\forall t \bullet ((t_1 \leq t < t_2) \wedge (occurs(In, t))) \Rightarrow was\_at(G3, t))$

Liveness Property: When the last train leaves the crossing, the gate eventually reopens.
Formally, $(was\_at(S_4, t_1) \wedge TCvar2 = 0) \Rightarrow \exists t \bullet (was\_at(G_1, t) \wedge (t - t_1 > 0))$
Note: The safety and liveness properties must be a logical consequence from the object models operational semantics and system synchronization semantics.

Chapter 3

Fundamental Issues in Software Measurement

In this chapter a survey of formal approaches to software measurement is presented. The underlying theories upon which fundamental measures are constructed, are: theory of real numbers, graph theory, information theory and category theory. For each theory the set of axioms and examples of fundamental measures are given. Measurement models and measures of software quality reviewed in this survey have a bearing on the research directions pursued in this thesis.

3.1 Categories of Software Measures

There are three categories of software measures: *fundamental*, or *direct measures* that reflect and directly characterize the empirical properties of the attribute in the numerical system; *indirect measures* which require first one or more fundamental measures of one or more attributes, than their combination (using some mathematical model) to measure (indirectly) another supposedly related attribute; and *conjoint measures* that measure the attribute and its components simultaneously.

3.1.1 Fundamental measures

The *fundamental measures* directly characterize the empirical properties of an attribute in terms of a number or symbol. The theoretical approach to measurement

described in [W97] allows the construction of theoretically valid fundamental measures. The author suggests to look for required scale types, then to select the requirements for the empirical structures and to construct one that is likely to provide the right kind of measure. Whitemire generalizes his approach introducing the notion of *software measure life cycle*. The author traces four phases for software measures development in analog with the software engineering well-known development phases *requirements, analysis, design, implementation.*

3.1.2 Indirect and Conjoint measures

The *indirect* measurement requires first one or more fundamental measures of one or more attributes, than their combination (using some mathematical model) to measure (indirectly) another supposedly related attribute. The *conjoint* measures measure the attribute and its components simultaneously.
There are two types of mathematical models of indirect and conjoint measures: *attribute models* (one external attribute is represented by one or more internal attributes), and *relationship models* (one attribute is defined as mathematical function of a list of other attributes).

3.1.3 Static and Dynamic Measurement

We can categorize the measurement as static or dynamic depending on the nature of the properties to be measured.
The *static measurement* is the evaluation of the set of static properties of the object, which assists the decision-making during the development process. The static measurement of the quality attributes can help decide whether the high-level system description, or a system design is better than another and choose the one that best meets the system goals.
The *dynamic measurement* requires the executing of the program before the metrics are applied. The dynamic measurement needs "probifacts" to be inserted within the software (for example, RTTI is the C++ future that allows to measure dynamically). The probifacts are reducing the speed of software execution, thus are to be removed (without damaging the system) later. Some of the metrics probes are required to stay within the system permanently. Thus, the probes should be chosen small, so

that their impact on the speed and the reliability would be invisible. The OO dynamic measurement evaluates the dynamic behavior (state-dependent response of a system/object to an external/internal event) of the system/object.
The *dynamic measurement* at design phase evaluates the dynamic behavior (state-dependent response of a system/object to an external/internal event) of the system/object. The dynamic measurement assesses the dynamic analysis of the object-oriented design (i.e., the design behavior when the implementation is executed) before the construction of code.

3.1.4 Global and Local Measurement

The measurement can be categorized as global or local in dependence of the scope of our measures. *Global measurement* is a measurement applied to a system as whole. *Local measurement* is a measurement applied only to a particular system's component.

Scope of Application

The measures can be categorized according to their scope of application following the corresponding Fenton et al. (1997) list:

- Structural and Complexity Metrics
- Quality Models and Metrics
- Productivity Models and Metrics
- Reliability Models and Metrics
- Cost and Effort Estimation Models and Metrics
- Performance Evaluation Models and Metrics
- Capability - Maturity Assessment Models and Metrics

The structural and complexity measures have to characterize directly the software properties in order to assess the development process. The object-oriented properties that can be measured are *coupling between objects*, *inheritance hierarchy*, *class*

internal coherence and *application size.* The conventional software properties that can be measured are the *control-flow*, *data-flow* and the *application size.* The choice of an empirical relational model depends on the underlying software development paradigm (OO, conventional), software abstraction and on the purposes of the measurement process.

3.1.5 Software Abstractions and Fundamental Measures

The abstractions of real-word software entities reported in the literature that allow the quantification of the software properties, can be classified as follows: *collection of tokens*, *control graphs and data dependency graphs*, *state spaces*, *object-predicated tables*, *covariance matrices* and *categories.*
The different abstractions of software are rooted either in the *theory of numbers* (collection of tokens), *graph theory* (Control graphs and data dependency graphs, State spaces), *information theory* [Sh69] (Object-predicated tables, Covariance matrices), or the *category theory* introduced as useful general structure for studying software in [W97] (Categories). All the above theories are founded in discrete mathematics. According to the theory on which the model of software is based, the software fundamental measures can be categorized as follows:

- Measurement based on the theory of numbers
- Measurement based on the graph theory
- Measurement based on information theory

The basics of mathematics necessary to understand the measurement of software are summarized in the following section.

3.2 Representational Approach

The software measurement is used to quantify objectively the abstraction of the software entities. To quantitatively characterize some attribute or property of a class of empirical objects, a model (abstract representation) of an object is necessary.
The aim of this section is to establish the basics of the theory of measurement, and to give the theoretical tool for building theoretically valid measures.

The purpose of the software measurement is to define clearly and unambiguously the population, the characteristics of software to be measured (domain representation model), and then to devise appropriate measures for these software characteristics together with instruments with which to measure them [HS96]. In order to define the domain representation model, the following must be identified: the set of empirical entities to be measured, their attributes to be measured, and a model for each attribute appropriate for the current environment.

Generally accepted *representational approach* to software measurement [FP97] consists in building numerical representations of the empirical observations in a numerical structure. The measure is a function that maps an empirical relational structure onto a numerical relationship structure. This mapping is known as fundamental measurement and leads to *fundamental measures*, or *direct measures*, because it directly characterizes the empirical properties of the attribute in terms of a number or symbol.

The fundamental measurement is formally defined as *a process of a structure preserving mapping* Φ *between a model of the empirical relationship in the form of an empirical relational structure* E*, and a model of the numerical relationship in the form of a numerical relational structure* N.

The *structure preserving mapping* is constrained by two theorems: *representation theorem* and *uniqueness theorem*. The representational theorem, also called the *representation condition*, sets forth the conditions (axioms) to be satisfied by the empirical relational structure, and shows how to construct the representation of an empirical structure into a numerical structure. The uniqueness theorem (or *uniqueness condition*) defines the mathematical transformations between different representations.

The *representation theorem* and the *uniqueness theorem* are framed by the conditions (*axioms*) imposed on an empirical structure by a representation in numbers/symbols. There are three classes of such axioms:

Class of necessary axioms. These are mathematically necessary conditions due to the representation being constructed. They lead to a set of allowable structures.

Class of structural axioms. These are unnecessary conditions that limit the allowable structures to a more manageable set.

Archimedean axiom. Every strictly bounded standard sequence is finite (based on

the Archimedean property of real numbers).

The triple $< E, N, \Phi >$ is also called a **scale** [FP97]. In general, the measurement theory involves the mathematical description of scales, measures, and methods of measuring.

3.2.1 Scales

Fenton and Whitemire ([FP97], [W97]) present five scale types in order of strength. Each type is defined in terms of its defining mathematical relations and its allowable transformations. The scale types are:

Nominal Scale

Real-word entities must be countable. They are assigned to numbers/symbols as a form of *classification*. The real meaning of using nominal scale is not measurement but classification. The defining relation is *equivalence* and partitions the set of entities into a set of scale values. The properties of the equivalence relation are *reflexivity, transitivity and symmetry*. Valid transformations are all isomorphisms of the form $x \rightarrow f(x)$ which preserve this partitioning.
Appropriate Statistics: Some robust techniques (include frequency, mode, contingency coefficients).
Appropriate Statistical Tests: Non-parametric tests which do not depend on ordering.

Ordinal Scale

The empirical entities must be countable. An empirical structure must have an empirical ordering relation $\succeq$ which is transitive and connected. The defining relations are *equivalence* and *greater than* $\geq$. Valid *transformations* are all isomorphisms of the form $x \rightarrow f(x)$ that are strictly increasing. The empirical ordering relation is a *weak order* and has the following properties:

1. $a \succeq b \Longleftrightarrow \Phi(a) \geq \Phi(b)$
2. $a \succeq b \vee b \succeq a$ (Connectivity)
3. $a \succeq b \wedge b \succeq c \Longrightarrow a \succeq c$ (Transitivity)

Appropriate Statistics: Robust techniques (including frequency, mode, contingency coefficients, median, quartiles, upper fourth, lower fourth, box length, upper and lower tails).
Appropriate Statistical Tests: non-parametric tests - Kendall's rank correlation coefficient τ, the Spearman rank correlation coefficient r_s, Kendall's ω.

Interval Scale

The mapping is preserving an empirical ordering relation $\succeq$ and the *ratio of intervals between objects*. The defining relations are the *ratio of any two intervals*, $\geq$ and *equivalence*. Valid *transformations* are any linear transformations of the form $g = ax + b(a > 0)$ that preserve order and intervals. The empirical ordering relation is a *weak order* and has the following properties:
Valid *transformations* form the *positive affine group*, if the domain of the numerical structure is real numbers and form the *power group*, if the domain of the numerical structure is the positive real numbers.

1. $a \succeq b \Longleftrightarrow \Phi(a) \geq \Phi(b)$
2. $If \quad c \succeq d \wedge a \succeq b \wedge (a \succeq b \Longleftrightarrow \Phi'(a) \geq \Phi'(b), \quad then$

$$\frac{\Phi(a) - \Phi(b)}{\Phi(c) - \Phi(d)} = \frac{\Phi'(a) - \Phi'(b)}{\Phi'(c) - \Phi'(d)}$$

Appropriate Statistics: Parametric statistical methods (including all the appropriate for an ordinal scale plus arithmetic mean, standard deviation, variance).
Appropriate Statistical Tests: all parametric and non-parametric tests.

Ratio

The mapping is preserving an empirical ordering relation $\succeq$, the ratio of intervals between objects and the *ratio of scale values*. The defining relations are *equivalence*, *ratio of any two intervals*, $\geq$, *concatenation* and *ratio of any two scale values*. The minimum scale value exists and it is a null (zero point). This scale permits calculation based on ratio and percentage. Valid *transformations* are of form $g = ax(a > 0)$, when $x \in$ *Set of real numbers*, or of the form $g = ax + b(a > 0)$ when $x \in$

Set of positive real numbers. The empirical ordering relation has the following properties:
I Valid *transformations*: form the *translation group* and *similarity group*,, if the domain of the numerical structure is all real numbers; of form the *similarity group*, if the domain of the numerical structure is the positive real numbers.

1. $a \succeq b \Longleftrightarrow \Phi(a) \geq \Phi(b)$
2. $a \sim nb \Longleftrightarrow \Phi(a) = n\Phi(b)$
3. $a \circ b \sim c \circ d \Longleftrightarrow \Phi(a) + \Phi(b) = \Phi(c) + \Phi(d)$

The defining relation is

$$\frac{\Phi(a)}{\Phi(b)} = \frac{\Phi'(a)}{\Phi'(b)}$$

I *Appropriate Statistics*: Parametric statistical methods (including the geometric mean and coefficient of variation, and those appropriate for the interval scale).
Appropriate Statistical Tests: all parametric and non-parametric tests.

Absolute

This scale presents counts of objects, cardinality of sets, and probabilities. The defining relations are all in the ratio scale plus the existence of the *unit of measure*. The valid *transformation* is the *identity* $x = f(x)$.
Appropriate Statistics: all the appropriate for the ratio scale.
Appropriate Statistical Tests: all the appropriate for the ratio scale.
To determine the relationships between attributes of the same or different types, *relationship statistical tests* should be applied.

3.2.2 Relationship statistical tests

There are two **classes** of relationship tests:

1. *Robust/Non parametric statistical methods* - used for nominal and ordinal scale measurements.
2. *Parametric tests* - used on data from the interval and ratio scales.

The measurement data should be gathered in order to be statistically analyzed. The scale type determines the type of statistics and statistical tests to be applied.

3.2.3 Basic Procedures of Measuring

In case the measurement is allowable, Whitemire [W97] outlined three basic procedures of assigning numbers to objects: *Ordinal Measurement*, *Extensive Measurement/Counting of Units* and *Solving Inequalities*.

Ordinal measurement procedure's first step is to determine an ordering relation for the objects, and the second is to assign a number sequence that preserves the order of the objects. The set of allowable transformations of the mapping of one representation to another is any transformation that preserves the order.

Extensive Measurement/Counting of Units procedure's first step is to select a unit and the second is to construct a standard sequence using the concatenation operation until the approximation of the length of the object being measured. The procedure depends upon an empirical concatenation operation and the construction of a standard sequence of objects. The set of allowable transformations can be obtained by applying the conversion factor between units.

Solving Inequalities procedure is used when it is impartial or impossible to construct a standard sequence. The first step is to write a set of inequalities reflecting the relationship between the entities, and the second is to find solution to this set. Any simultaneous solution to the set of inequalities is a valid numerical representation.

3.3 Validation of Software Measures

From the theoretical point of view, the *formality* is a must when defining a measure. Any measure should be developed and tested in the context of measurement theory in order to clarify whether any specific measure is appropriate to be applied in specific situation. The measure should be theoretically grounded, with at least a weak order and expressed in some unit system.

Validating a software evaluation measure is a process of ensuring that it is a proper numerical characterization of the claimed attribute. Validating a software predictive metric is a process of establishing the accuracy of the prediction by comparing model performance with known data. There are two types of software metrics validation:

theoretical and *empirical.*

The *theoretical validation* is a process of ensuring that the fundamental measure is satisfying the representation condition of measurement theory.

The *empirical validation* is a process of establishing the accuracy of the software measurement by empirical means. In case of *assessment, comparison, investigation purposes measurement*, the empirical validation identifies the extent to which a measure characterizes a stated attribute by simple test against reality. In case of *prediction, estimation purposes measurement*, the empirical validation formulates a hypothesis about the prediction and then experimentation to test the hypothesis by comparing the model performance with known data (controlled empirical experiment). To formulate the null hypotheses necessary for validating the predictive abilities of the measures, the prediction system should be classified as:

- Using internal measures of early life-cycle products to predict internal/external measures of later life-cycle products.
- Using early life-cycle process/resource attribute measures to predict measures of later development phases products/resources attribute measures.
- Using process measures to predict later process attribute measures.
- Using internal product measures to predict process attribute measures.

Once the hypothesis has been confirmed, the cause-and-effect relationships between the measures have to be identified. The most used statistical test is *correlation*. Correlations can be run between two or more attributes and measures the extent to which the value and direction of change in one attribute is tied to that of another attribute. When the relationship between two or more attributes is established, the *regression analysis* is applied to determine what that relation might be (linear, polynomial, or exponential). It is often the case that a single measure, or even a number of measures, do not adequately characterize a software entity due to the *collinearity* problem (each measure contributes the same or nearly so characteristic that can lead to incorrect conclusions on measurement data). The *principal component analysis* was developed to combat this problem. The method uses a given set of data to select and weight the principal components, which amounts to built-in correlation. The principal component analysis is the method for creating indirect measures from collections of direct

and other indirect measures [W97]. Once the cause-and-effect relationships have been identified, a useful and valid predictive (prognostic) model can be created.

3.4 Measure Construction: Basic Development Steps

A measure has a life-cycle similar to the software life-cycle. The *first step* in the theoretical validation process is to define requirements for measurement. The requirements are the constraints of the chosen scale type, and the structural requirements of a potential mathematical model (representation) of a measure.
The *representation condition* sets forth the requirements to be satisfied by the mathematical model of a measure.

1. **Determine your requirements for measurement** *(define requirements)*

   - Requirements are obtained from the chosen *type of scale.*
   - The scale type is set by the choice of *kind of analysis* to perform on the data.
   - The kind of analysis depends upon the *role (the use)* for the technical measurement:
     - *Estimation:* requires at least the interval scale.
     - *Prediction:* requires at least the interval scale: would work better with ratio or absolute scale measures.
     - *Assessment:* requires at least nominal scale.
     - *Comparison:* requires at least ordinal scale.
     - *Investigation:* requires at least nominal scale.
   - The process results in a set of candidate mathematical structures available for the above requirements. If the set is empty than the requirements have to changed.

2. **Analyze the model** (*analyze the requirements*).

   The approach is to take a mathematical structure, identify its requirements, and see if the elements, relations, and operations required by the structure are found in the model. Eliminate any structure for which you cannot meet the requirements.

3. **Select the target structure** (*design a candidate implementation*)

   Select a structure that requires less effort to collect measurement data or to calculate the measure, and is most closely aligned with the goals.

4. **Construct a representation** (*build an application*)

   (a) Map the structure elements to numeric elements; the empirical relations and operations to numeric relations and operations.

   (b) Select a unit.

   (c) Validate the representation (*testing*).

The *second step* is to construct the representation such that it satisfies the measurement requirements. A mathematical proof is required to show that the measure satisfies its requirements.
When the theoretical validation of the measure is completed, it has to be empirically validated (i.e., the representation has to be tested).

Testing the Representation

The collected measurement data has to be used to show that the measure satisfies the following criteria:

1. *Tracking:* the extent to which changes in the value of the measure follow changes in the value attribute.

2. *Consistency:* the extent to which the value of the attribute and the value of the measure share rankings when measurement data is sorted in order:

   $a_1 \preceq a_2 \ldots \preceq a_n \Longleftrightarrow \Phi(a_1) \leq \Phi(a_2) \ldots \leq \Phi(a_n)$

The process results in an empirical relational structure which is guaranteed (mathematically proven) to map to a real numerical structure with the properties of a particular scale type.
The empirical validation process is illustrated on different configurations of the Train-Gate-Controller systems case study.

3.5 Software Measurement Based on the Theory of Numbers

The abstractions of real-world entities that allow the quantification of software properties as a collection of tokens, are rooted in the theory of natural numbers.

3.5.1 Collection of Tokens

Collection of tokens means static counting of elements in a set. Tokens are objects to be collected and could one of the following: lines of code, operators, operands, objects, classes, or functions derived from the source [W97]. The procedure of collection of tokens is based upon absolute scales of counts of tokens. Therefore, the mapping is on the set of natural numbers, and the measures are easily collected on the ratio and absolute scales.

Axiomatic Approach The set of axioms imposed on an empirical relational model by the properties of the natural numbers, is as follows:

Axiom 1. Every nonempty set has a minimum element

Axiom 2. Archimedeum axiom

Axiom 3. There is at least a weak order between the elements of a set

Axiom 4. Commutativity: $\forall a, b \in S : \{a + b = b + a\}$

Axiom 5. Associativity: $\forall a, b \in S : \{a + (b + c) = (a + b) + c\}$

Axiom 6. Exists a unity: $\forall a \in S : \{a \times 1 = a\}$

Axiom 7. $\forall a, b, c \in S : \{a + c = b + c \Rightarrow a = b\}$

Axiom 8. $\forall a \in S : \{\exists succ(a) \bullet succ(a) \neq 1 \& (succ(a) = succ(b) \Rightarrow a = b\}$

3.5.2 Measures

Fundamental measurement of the physical size of a software is based on the counting of tokens.

Whitemire ([W97]) is refining the static measurement of the physical size of OO product in *population* (counting of elements in a set of objects) and *functionality* (counting of elements in a set of relationships between the elements of a system).

Population and Functionality

Axiomatic Approach The following set of axioms for the fundamental measurement of the population $pop(A)$ are defined:

Axiom 1. Population is nonnegative: $pop(A) \geq 0$

Axiom 2. Population can be null: $A = \emptyset \Rightarrow pop(A) = 0$

Axiom 3. Population is additive for disjoint populations: $A_1 \cap A_2 = \emptyset \Rightarrow pop(A_1 \cup A_2) = pop(A_1) + pop(A_2)$

Axiom 4. Population follows the siege principle: $pop(A_1 \cup \ldots \cup A_n) = \sum_{i=1}^{n} (-1)^{i-1} \alpha_i$ where α_i is the sum of the populations of the intersections taken i at a time.

Axiom 5. Population is monotonic increasing: $A_1 \subseteq A_2 \Rightarrow pop(A_1) \leq pop(A_2)$

Axiom 6. Population of two merged populations cannot exceed the sum of two

Axiom 7. Population forms a weak order

The functionality is defined by Whitemire as "the set of functions derived from the source" (i.e., the set of functions requested, the set of functions delivered).
Example of functionality measurement are the OO coupling direct measures defined in the corresponding section.

Axiomatic Approach

Whitemire ([W97]) denotes the functionality as $fun(A)$ and lists the properties that are similar to those for population.

Axiom 1. Functionality is nonnegative

Axiom 2. Functionality can be null

Axiom 3. Functionality is additive for disjoint sets of functions

Axiom 4. Functionality follows the siege principle

Axiom 5. Functionality is monotonic increasing

Axiom 6. Functionality of two merged populations cannot exceed the sum of two

Axiom 7. Functionality forms a weak order

An early attempt to measure the size of a source code for an imperative language based on counting of tokens has been proposed by Halstead [H75]. Halstead defined the abstraction of a program as a collection of operators and operands. Halstead's fundamental software science measures for these tokens were μ_1 (number of unique operators), μ_2 (number of unique operands), N_1 (total occurrences of operators), and N_2 (total occurrences of operands). Fenton argued that only three of the Halstead's measures, namely the *length* $N = N_1 + N_2$, *vocabulary* $\mu = \mu_1 + \mu_2$ and *volume* $V = N \times \log \mu$ are theoretically valid measures of internal code attributes that reflect different views of code physical size.
We can define the size of S as the cardinality of a set of objects $\{O_1 \ldots O_n\}$. The traditional length code measure is the number of lines of code LOC. Fenton et al. [FP97] review different approaches for defining a token "line" in dependency on a particular code measurement purpose. The example of industrial use of length measuring are the COCOMO (Constructive Cost Model [Bo81], based on the measuring of delivered source instructions DSI or source lines of code $SLOC$.
The OO measures of counting of tokens are:

- MOOSE metric *Number of attributes (NOA) [CK93]* defined as $NOA(C_i) = m$
  .
- MOOSE metric *Number of public methods (NOM) [CK93]:* $NOM(C_i) = n$.
- *Number of responsibilities the class has (NOR):*
  $NOR(C_i) = NOA(C_i) + NOM(C_i)$
- *Lines of code per class (LOCC):*

$$LOCC(C_i) = \frac{\sum_{j=1}^{n}(lines\ \ of\ \ code\ \ for\ \ M_j)}{n}$$

- *Number of parameters per visible method (NPS):*

  $NPS(m_j^i) = number\ of\ parameters\ admitted\ by\ \ M_j\ of\ class\ C_i$

- Li and Henry ([LH93]) metric *Data Abstraction Coupling* DAC defined as DAC = *Number of Abstract Data Types Defined in a Class* is a measure of the coupling complexity due to declaration of variables of Abstract Data Type within the class. *Range of metric's values* is $[0, N_{DAC}]$. Higher value indicates higher level of coupling complexity and higher maintenance effort.

- Li and Henry ([LH93]) Number of Methods in a Class NMC is a class interface increment metric. It is defined as NMC = *Number of Local Methods in a Class.* The *range of metric's values* is $[0 \ldots N_{NMC}]$. Higher value indicates more complex class' interface and higher maintenance effort.

Measurement of Polymorphism in OO System

Notion of Polymorphism An operation is polymorphic when its semantics depend on the context of invocation. Polymorphism allows the implementation of a given operation to be dependent on the object that "contains" the operation: a subclass should minimally adhere to what the superclass prescribes, but it is allowed to define an operation differently as long as they conform to what the superclass prescribes (i.e., it may strengthen a postcondition of a transition in a superclass [Ch97]. Benlarbi and Melo [BM99] are classifying the polymorphism in dependency on the encapsulation level as class level polymorphism and system level polymorphism. At class level the polymorphism is *pure* (applying a single method to arguments of different types in the same context). At system level the authors consider two forms of polymorphism: *method overriding* (the behavior described in a parent class is altered in the descendant class) and *method overloading* (or *ad hoc*) (the same name denotes different methods). Benlarbi and Melo [BM99] define the use of the same name and the same signature in an overridden method as *dynamic polymorphism*, and the use of the same name but different signatures in different classes (linked or not by inheritance) as *static polymorphism.*

Examples of Existing Measures

Benlarbi and Melo [BM99] define the *pure* polymorphism measure named *Parametric Overloading OVO*. *OVO* is simple counting of number of times when "the same method is envoked with the same name with different signatures inside the scope of the same class". It is a static fundamental measure of OOD on the absolute scale.
The *method overriding* polymorphism measures have to be derived from the inheritance hierarchy tree abstraction. The basic characteristics of the tree abstraction may be used as fundamentals measures of polymorphism. Thus, the static fundamental measure of the scope (size) of polymorphism may be defined as the cardinality of a set of overridden methods. The measure of the polymorphism in a descending inheritance line is the *fan-down* metric defined as cardinality of a set of subclasses that redefine any feature. The MOOD's metric *Polymorphism Factor POF* [BaPS98] is quantifying the ratio of a number of overriding/redefined methods and total number of methods in a descending inheritance line. Benlarbi and Melo [BM99] static and polymorphism measures *SPA (Static Polymorphism in Ancestors)*, *SPD (Static Polymorphism in Descendants)*, *DPA (Dynamic Polymorphism in Ancestors)*, *DPD (Dynamic Polymorphism in Descendants)* are calculating the cardinality of a set of static/dynamic polymorphism function members within the set of distinct ancestors/ descendants of a class.
The *method overloading* polymorphism can increase the complexity of a system if the semantic consistency across the interfaces of the objects is not provided.

Axiomatic approach The mathematical notion of *function overriding* is used to model polymorphism in software. The notation of *f is overridden by g* is $f \oplus g$.
Any operation/method can be seen as a function (deterministic relation) that maps a particular element from an input set (domain) onto the output set (codomain). The domain and codomain are described by the pre- and postconditions. Lets consider a polymorphism of two methods f and g such that g is overriding/ overloading f. A polymorphism in software can be interpreted as choosing between different functions producing different results for some common domain element (static binding and dynamic binding). The mathematical terming for this procedure is *function overriding*. The domain of a method resulting from method overriding has to be $(dom(f) \bigcap dom(g)) \setminus dom(f)$. The following axioms for polymorphism can be stated

based on function overriding properties:

Axiom 1. *Polymorphism(f,f)=f* (idempotentcy)

Axiom 2. *Polymorphism(f, Polymorphism(g,h))= Polymorphism (Polymorphism(f,g),h)* (associativity)

Axiom 3. *Polymorphism*$(\emptyset, g) = g$ (has left unity)

Axiom 4. *Polymorphism*$(g, \emptyset) = g$ (has right unity)

Axiom 5. $dom(f) \bigcap dom(g) = \emptyset \Rightarrow$ Polymorphism(f,g)= Polymorphism(g,f)

Benlarbi and Melo [BM99] claim that the some forms of polymorphism may decrease the reliability of the OO software.

Measures Based on State Spaces

The dynamic measurement assessing the development of the object-oriented design during its dynamic analysis depends on the number of the objects' behaviors and the different transactions required to solve the application. An object's *behavior* is concerned with object's state changes (internal behavior) and *message passing (request from another object to perform a service)* (external behavior). The message passing is linked to the state changes through mechanism called *binding* [W97]. Each transaction consists of the operations *message passing*, *message binding* and *object's state transition(s)*.

Let us consider an application A and a subsystem S that solves A. The system S can be constructed as a collection of objects $\{O_1, \ldots O_n\}$ that interact among themselves in solving the problem A. Therefore the state spaces abstraction is based on the graph theory. The state spaces abstraction is useful when evaluating the dynamic behavior (state-dependent response of a system/object to an external/internal event) of the system/object. The particular in the set membership of state spaces is that the time factor is included: set membership have to be determined at a specific point of time or a time interval.

The state spaces abstraction is useful when evaluating the dynamic behavior (state-dependent response of a system/object to an external/internal event) of the system/object.

Fundamental Measures for Dynamic Analysis

Fundamental measures are the *volume* of the state space, the *number of different transactions* and the *number of events* to which it need to respond.
Let k be the number of different transactions dependent only on an application A; $\{T_1, \ldots T_k\}$ be the set of different transactions required to solve the application A; $m_i, i \in [1, k]$ denote the number of messages exchanged to complete the transaction T_i; t_i be the time of completing the transaction T_i (in units); t_{i_j} - the time (to complete the execution of message-passing, binding and state transition(s)) calculated for the j^{th} message ($j \in [1, m_i]$) from the dynamic model of S.
The following dynamic fundamental measures are proposed:

k (the cardinality of the set of different transactions): fundamental measure of the functionality of S.

$m = \sum_{i=1}^{k} m_i$ (the cardinality of the set of messages exchanged to complete all the transactions): fundamental measure of the functionality of S.

$t_i = t_{i_1} + t_{i_2} + \ldots + t_{i_{m_i}}$ (the time of completing the transaction T_i (in units)). t_i is a measure of the relative contribution (weight) of the transaction T_i for the total functionality of S.

$t = \sum_{i=1}^{k} m_i$ (the time of completing all the transactions T_i (in units)). t is a measure of the factor time in the functionality of S.

Axiomatic Approach The volume is equivalent to the population measure (the cardinality of a set of objects) at a specific point of time or a time interval [W97]. The number of different transactions is equivalent to the functionality measure (the cardinality of a set of relationships) determined at a specific point of time or a time interval. When the time factor is held constant, the Whitemire axioms respectively for population and functionality hold.

3.6 Graph Theory Based Software Measurement

From the mathematical point of view graphs are algebraic structures, based on a set and relationships between elements of the set. Since graph theory is well developed, a

graph representation of software allows to formally define software characteristics in terms of well known characteristics of graphs, and to quantify them directly applying fundamental measurement.
Most graphical representations of software such as control-flow, data-flow models, entity-relationship models, state transition models, class models and use case models can be considered as a graph.
The unified approach to abstract the control - flow structure of the structured program, or low-level design is to use *flowgraphs.*

Flowgraphs

A program is considered to be structured if it is implemented using only a small set of allowable constructs. The optimum set of constructs consists of *sequence*, *selection* and *iteration.*
Flowgraph is a directed graph in which two vertices, the start and the stop, have special properties: the stop vertex has outdegree zero, and every vertex lies on some walk from the start to the stop [FP97].
There are two operations defined on the set of flowgraphs: *sequencing* **";"** and *nesting* **"()"**. Given two flowgraphs F1 and F2, their sequencing produce a new flowgraph (F1; F2) by identifying the stop vertex of F1 with the start vertex of F2. Given two flowgraphs F1 and F2, their nesting produce a new flowgraph (F1(F2 on v) by replacing the edge leaving the vertex v, with F2. The flowgraphs that cannot be decomposed non-trivially by sequencing and nesting are named *prime flowgraphs* .
Each flowgraph is associated with the unique *decomposition tree* that describes how the flowgraph is built by sequencing and nesting primes. The *prime decomposition theorem* asserts that every flowgraph has a unique decomposition by sequencing and nesting into a hierarchy of primes, i.e. non-decomposable flowgraphs [FP97].
The most measures based on graph theory are measuring code structure complexity. Set of axioms have been proposed by Bache (1990) and cited [FP97] for complexity measures of general control structure.

Axiomatic Approach Let $F, F1, F2, \ldots, F_n, G, H$ be prime flowgraphs, μ be a measure of general control structure. The above axioms are:

Axiom 1. $\mu(F;G) > \max\{\mu(F), \mu(G)\}$

Axiom 2. $\mu(F;G) = \mu(G;F)$

Axiom 3. $\mu(H) > \mu(G) \Rightarrow \mu(F;H) > \mu(F;G)$

Axiom 4. $\mu(H) > \mu(G) \Rightarrow \mu(F(H, F2, \ldots, F_n)) > \mu(F(G, F2, \ldots, F_n))$

Axiom 5. $\mu(F(G)) > \mu(F;G)$

Axiom 6. $\mu(H(G)) > \mu(G(H))$

Control-flow measures based on flowgraphs

The classic *McCabe's cyclomatic complexity* measure [Mc76] for the flowgraph F is calculating the complexity in terms of the number of linearly independent paths through

$$F: \ \nu(F) = (number\ of\ edges) - (number\ of\ vertices) \ + \ 2$$

The measure $\nu(F)$ is objective indicator of testability and maintainability of structured software, but cannot be considered as valid complexity measure because it does not satisfy the Bache's axioms 5 and 6.

Fenton et al. [FP97] provides hierarchical measures for software structuredness based on his definition of the set of basic $S - graphs$ and $D - graphs$ (legal control structures suited for particular applications). The *McCabe's essential complexity* measure $e\nu(F) = \nu(F)$ - number of D-structured primes subflowfraphs is proposed to capture the overall level of structuredness in a program.

Detailed *minimum number of test cases* measurement for the structural testing approach is developed in [FP97] applying the prime decomposition theorem. The definition of a measurement formula of calculating the minimum number of test cases depends on each testing approach.

In object oriented software, the flowgraph-based structured measures are applicable to measure the static internal structural complexity of methods within a class. The class-level calculation for a class with m methods, each of cyclomatic complexity ν_{ij}, is: $S_i = \sum_{j=1}^{m} \nu_{ij}$. In more general case, when c_i is the static complexity of a $method_i$, the class-level complexity is calculated by the Chidamber and Kemerer's metric $WMC = \sum_{i=1}^{m} c_i$ [CK93]. The average complexity measurement at module

level is useful because the algorithms may be self-contained or may reference other algorithms within the class:

$$Average\ Complexity = \frac{\sum_{\#\ of\ services\ in\ class} \nu(service)}{total\ \#\ of\ services}$$

At interclass level, the analog to the McCabe's cyclomatic complexity is the *Association complexity measure* AC defined as $AC = A - C + 2P$, where A is the number of associations (edges) in the class diagram, C is the number of classes (vertices), and P is the number of disconnected parts.

Data-flow measures based on graph theory

The data-flow measures have to capture the level of the inter-modular or intra-modular data moving through a system [FP97]. The abstraction of the information flow is a *data dependency (oriented) graph*, where the vertices are software components (modules, classes, objects) and the edges correspond to the interchange of data between the software components. These measures are applicable during the early development phases.

The traditional software *Henry and Kafura's information flow measures* [HK81] consider the local direct flows, local indirect flows and the global flow (via global data structures in common use) of the structured design. *Fan-in* of a module M corresponds to the *indegree* graph characteristic and thus is theoretically valid. It shows the number of local flows that terminate at M, plus the number of data structures from which information is retrieved by M. *Fan-out* of a module M corresponds to the *outdegree* graph characteristic and thus is theoretically valid too. It calculates the number of local flows that start at M, plus the number of data structures updated by M.

The original *information flow complexity of a module* M measure is calculated as

$$length(M) \times ((Fan - in(M) \times (fan - out(M))^2$$

This definition is not theoretically valid from the theoretical point of view because *length* and *in - outdegree* are not orthogonal characteristics of the underlying abstraction. Shepperd [Sh90] has refined the definitions of fan-in and fan-out, and modified the complexity measure disregarding the $length(M)$.

In object oriented software, measures *Fan-in* and *Fan-out* refer to the number of collaborating classes. The association and the aggregation relationships are counted during the OO Analysis and OO Design phases. High *Fan-in* value indicates good object designs and a high level of reuse. High *Fan-out* indicates excessively complex dependence on other modules [HS96].
The association and the aggregation relationships counting doesn't take into consideration the number of references made (statically or dynamically) to the collaborating class. Thus, it is irrespective of the amount of collaboration.

3.6.1 Application Size Measurement

The internal product quality attribute which can be measured statically and dynamically early in the life-cycle is the *size* of the software system. The discussion of the sources of size and its different aspects are given in [FP97] and [W97].
Fenton et al. [FP97] define three criteria for the factor size: length (physical size of the product), functionality (the quantity and quality of functions supplied by the delivered product or in a description of how the product is supposed to be) and computational complexity of the underlying problem.
Whitemire [W97] is refining the notion of length in three more precise views, namely *population* (static counting of elements in a set of objects), *length* (defined as static counting of elements in a chain of connected physically or conceptually elements) and *volume* (dynamic counting of elements in a set of objects). The author argued that the computational complexity of the underlying problem is not a static aspect of the size of an application, but dynamic design measure. Therefore, Whitemire defines four aspects of software size: *population*, *length*, *volume* and *functionality*. The functionality is defined by Whitemire as "the set of functions derived from the source" (i.e., the set of functions requested, the set of functions delivered) and means static counting of elements in a set of relationships between the elements of a system.

Axiomatic Approach

Population and functionality are based on the theory of natural numbers and are discussed in the section 3.5. *Volume* is a dynamic measure discussed in the section *Measures Based on State Spaces.*

Length measurement is defined as "the number of elements in a chain of connected elements" where the connections are physical (graph representation) or conceptual (tree representation). Whitemire [W97] lists the following axioms for length:

Axiom 1. Length is nonnegative

Axiom 2. Length can be null

Axiom 3. Length is monotonic non-increasing for connected elements

Axiom 4. Length is monotonic nondecreasing for elements in different chains

Axiom 5. Length of a set of disjoint (non-overlapping) chains is the maximum of the lengths of the member chains

3.6.2 Measures Based on Graph Representation of the OO Design

The quality of the design is essential for the economics of object oriented software development, therefore most of the publications on object oriented software metrics are dedicated to the quality of object oriented design. To make use of the measurement during the object oriented design process, the components of the design have to be measured *directly* in order to assess the course of the design. The Object Oriented Design properties that can be measured directly are: coupling between objects, inheritance hierarchy, cohesion and application size.

Measurement of Coupling in the OO System

Notion of Coupling. The abstraction used to model the coupling is a directed graph. Coupling is an internal product attribute that describes the nature and extend of the logical/ physical connections between the components of a system.
There are two general classes of coupling: *necessary* (required to support interaction among components of the system); and *unnecessary* as result of bad design (results in low modifiability of a system). A system may be a component in a larger system, in which case it is called '*subsystem*. The IEEE standard [IEEE90] defines coupling as " a measure of the interdependence among modules in a computer program".

Coupling is defined in [W97] as measure of strength of the physical connections between components of a system.
The only physical connections which may exist *between classes* are instances of the relationship types defined in the formal object model: inheritance, association, aggregation. *Inheritance coupling* is the degree to which a derived class uses inherited attributes and methods. *Interaction coupling* is the degree to which information in a message between two objects is used by the receiving object. *Abstract coupling* is presented when a class is dependent only on the type or interface of the abstract class, not on the implementation of its properties (attributed or behaviors). This form of coupling is helpful when avoiding compile-time dependencies between classes in a design.
The coupling *within a class* is manifested by the connections between the methods of a class and its state information. The methods can be coupled to each other when they access directly the attributes of a class. Any time one method in a class calls another method in same class, the two methods are coupled.
The three kinds of coupling identified in [LH93] are

- coupling through inheritance,
- coupling through message passing, and
- coupling through data abstraction.

This classification differs from the classification proposed in [W97]:

1. **Interface coupling** Whenever an object refers to another object through its interface methods, interface coupling is said to exist between the two objects.
2. **Inside internal coupling** This is the result of coupling of the methods to the object's attributes, or of the methods of the component object's attributes that make up the a composite object.
3. **Outside internal coupling** This occurs due to the knowledge that one class has of another. There are two forms of outside internal coupling:
    - One of a pair of otherwise unrelated objects may access internal, even private, attributes or functions of the other object.

- A specialization accesses attributes and methods of its generalization in a way other than through the generalization public interface.

enditemize

Axiomatic Approach. Whitemire ([W97]) is summarizing the theoretical approaches to the notion of coupling and lists the following axioms for the coupling of a design component C ($cu(C)$, where the underlying abstraction of a design is graph $G =< V, E >$:

Axiom 1. Coupling is nonnegative

Axiom 2. Coupling can be null

Axiom 3. Adding an inter-component relationship does not decrease coupling

Axiom 4. Merging two components does not increase coupling

Axiom 5. Merging two unconnected components does not change coupling

Axiom 6. Coupling forms a weak order

Examples of Valid Measures. The following metrics illustrate a possible way of coupling direct measurement based on graph representation:

- the MOOSE [CK93] metric *Coupling between objects/classes*
- Li and Henry [LH93] metric *Message passing coupling* and the MOOD [BaPS98] *coupling* metric.

Measurement of Cohesion in OO System

Notion of Cohesion. Cohesion is an internal product attribute defined as "degree to which the tasks performed by an entity are functionally related", or, in other words, the degree to which the class describes a single abstraction. It can be measured directly in a *white box* fashion by examining measure's intervals.
The traditional view of (*structural*) cohesion defines it in terms of the connections between components of a class or a module. Chidamber and Kemerer [CK94] define

cohesion in terms of the intersections of instance variables used by the methods. The Chidamber and Kemerer's metric *Lack of Cohesion in Methods (LCOM)* [CK94] illustrates a possible way of direct cohesion measurement.
A different view on cohesion is to define it in terms of singleness of purpose (*semantic, or logical*), i.e., as the degree to which the entities (such as class, an operation, or a module) contribute to the carrying out of a single, identifiable purpose. The measurement of the following two forms of semantic cohesion may be carried out by examining the public interface of the design component:

1. The strength of relationship between the external properties of a component and the external properties of an abstraction it implements;
2. Contribution of the internal elements of a component to providing the external properties.

Axiomatic Approach. Whitemire ([W97]) defines the cohesion $coh(x, a)$ as a relation between the abstraction of a design component x and the abstraction of the corresponding domain object a, and is proposing the following axioms for cohesion measurement:

Axiom 1. Cohesion is nonnegative

Axiom 2. Cohesion is independent of size

Axiom 3. Cohesion can be null

Axiom 4. The cohesion is an external characteristic of a collection of components (independent from the internal structure of the components).

Axiom 5. Cohesion forms a weak order

Examples of Valid Measures. Whitemire is giving a theoretically valid measure of cohesion defined as:

$$coh(x, a) = 1 - \frac{| x \setminus a |}{| x |}$$

where $\mid x \setminus a \mid$ means the set of attributes in x that are not in the domain object a. The measure is on the absolute scale. *coh* reach its maximum (corresponds to the best case) when $\mid x \setminus a \mid = 0$.
A system measure of cohesion is given in [FP97] as:

$$cohesion\ \ ratio = \frac{\#\ \texttt{of modules having functional cohesion}}{\texttt{total}\ \#\ \texttt{of modules}}$$

Measurement of Class Inheritance Hierarchy in OO System

Notion of Inheritance. The inheritance principle allows the incremental definition of the classes by reusing previously defined classes as the basis for new objects. According to Henderson-Sellers [HS96], inheritance is an internal attribute of OOD that reflects the object-to-object dependency and class-to-class visibility requirements. The result of inheritance is a coupling between classes/objects. Inheritance can be considered as sharing of declarative properties such as attributes and constraints, and thus is rooted in the static measurement. There are two approaches to classifying inheritance: *single* (allows a subclass to have only a single parent class) vs *multiple* (a subclass may inherit from more than one parent class), and *specification* inheritance (a-kind-of relationship) vs *implementation* inheritance (or incidental that deals primarily with code reuse).
The underlying abstraction of the class inheritance hierarchy is a tree. The characteristics of the tree morphology (*size*, *length of path/walk*, *distance between two vertices*, *depth* of a tree, *height* and *width of a tree*) are theoretically valid definitions for the fundamental measures of the inheritance. Complete discussion of the inheritance measures is given in Henderson-Sellers [HS96].

Examples of Valid Inheritance Measures. The existing measures are: the Chidamber and Kemerer's metric *depth of inheritance tree (DIT)* [CK93], called also *nesting level* within the inheritance hierarchy; Chidamber and Kemerer's metric *Number of Children of a Class$_i$ (NOC$_i$)*; and the following listed in Henderson-Sellers [HS96] measures: Maximum depth of inheritance tree; Mean depth of inheritance tree; average *NOC* for the whole inheritance tree; number of distinct inheritance hierarchies within a system.
Inheritance is an essential future of the object oriented paradigm. Its use results in

coupling between classes/objects. Some authors argue that inheritance reduce the actual design complexity. Another existing opinion is that the use of the inheritance provides decrease of the maintainability level of the system.

3.7 Information Theory Based Software Measurement

Information theory based software measurement is used to quantify objectively the software complexity in terms of an amount of information based on some abstraction of the software. The information content of the software product is synonymous with the complexity of the product [KA94], so measuring the amount of information is assessing the quality improvement and the maintainability of the software system. The information theory based measurement is objective because it evaluates the abstraction of a software product independently of the software notation to be used and the life cycle phase to be applied in.

3.7.1 Notions of Information Theory

There two fundamental information theory notions: *Entropy* and *Excess-entropy*. *Entropy* is information theory measure of the average information of an event. Considering a set of n events and their probability distribution $\{p_1, \ldots, p_n\}$, the formula for calculating the entropy H is:

$$H(p_1, \ldots, p_n) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

The definition of the entropy of a k - partition of a set of n elements, where the subset$_i$ has n_i elements and $\sum_{i=1}^{k} n_i = n$, is:

$$H = -\sum_{i=1}^{k} \frac{n_i}{n} \log_2 \frac{n_i}{n}$$

The *maximum entropy* (when all events are equally likely) is $H = \log_2 n$. The unit of measure is *bit*. The basis for the probability distribution of the n possible events is the key modeling decision when measuring software complexity with entropy.

Excess-entropy is information theory measure of the interaction between subsystem components or similarities among the software products. The excess-entropy C is defined as the difference between the sum of the entropies of parts and the entropy of the whole. The formulas for the calculation of the excess-entropy C of a set of n random elements $\{el_1, \ldots el_n\}$ are:

$$C(el_1, \ldots el_n) = \sum_{i=1}^{n} H(el_i) - H(el_1, \ldots el_n)$$

$$H(el_1, \ldots el_n) = - \sum_{\textit{all combinations of } el_i \textit{ values}} p(el_1, \ldots el_n) \log_2 p(el_1, \ldots el_n)$$

The definition of the interactions among the elements is the key modeling decision when measuring software complexity with excess-entropy.

3.7.2 Measures

For the software complexity models that define an abstraction of a software product as a *set of objects partitioned into k equivalence classes* by some attribute, the entropy is interpreted as the average complexity of the elements of the set.

For the software complexity models that define an abstraction of a software product as a *graph*, the entropy can be interpreted as the average complexity of the graph. The Mowshowitz' *automorphism entropy* H_g measure of the symmetry of a graph, and *chromatic entropy* H_c measure of the information content of graph connections, can be used to calculate the information content of the graph.

When the software complexity model uses non-quantitative data discrete probability distribution, an *object-predicate table* is proposed to model such data, and an *excess-entropy* to measure the complexity of the interaction among objects via predicates in common use.

For the software complexity measurement modeled as continuous random variables from a multivariate distribution, the *covariance matrix entropy* represents the average complexity of system elements, and the *excess entropy* represents the correlation among the measurements.

Measures Applicable during the Specification Phase

The specification measures reviewed in Khoshgoftaar et al. (1994) are *Hellerman measure of software work w(f)*; *Paulson and Wand* application of $w(f)$ as a measure of a document's complexity; *Coulter, Cooper and Solomon problem space (PS) entropy* $H(PS) = \log|PS|$. The reviewed measures provide an evaluation of the complexity of alternative decompositions of the high level software system description into components.

The first two measures are based on the inputs and outputs of a function f as stated in the software specification. The last measurement model defines a problem space PS as all possible execution time output states allowed by the software specification. The measures reviewed provide an evaluation of the complexity of alternative decompositions of the high level software system description into components.

Measures Applicable during the Design Phase

The design measures reviewed in Khoshgoftaar et al. [KA94] are *Mohantly excess-entropy* measure of system structure complexity via shared information, applied to the design phase; *Lew, Dillon and Froward*'s system complexity measures based on control/data messages passing between components, applied during the detailed design phase (the Mowshowitz's approach of calculating entropy of graph is applied); *Lew*'s component complexity measure M_i and system complexity measure K defined as Euclidean norm of component complexity M_i. Alternative designs at comparable level of abstraction could be evaluated on the basis of Mohantly's design complexity measure.

Measures Applicable during the Implementation Phase

The implementation measures reviewed in Khoshgoftaar et al. (1994) are *Chen's control structure entropy* Z_n based on a control flow graph (counts of IF statements in a program code); *Berlinger's code complexity measure* M and *language entropy* H (the amount of information per token in the programming language subset used by the application); *Cook's measure for module complexity* M' based on Berlinger's measure M and applied to assembly language programs; *Harrison's module complexity measure* $AICC$ based on the distribution of operators within a program; *Davis*

and LeBlanc's entropy measures of the structure of chunk connections, content or size (*chunk* is a sequence of statements with only one In control statement at the top), based on equivalence classes; *Robillard and Boloix complexity measure* I_{PGM} based on the excess entropy calculation of the interconnections between statements in a program; *Khoshgoftaar and Munson's methodology* for combining primitive software measures into a synthetic system complexity measure γ, based on excess entropy; *Zhuo, Lowther, Oman and Hagemeister* empirical validation of γ with respect to *maintainability* (γ is based on the Halstead V; Extended McCabe cyclomatic complexity; Lines of Code; Lines of Comments; Number of statements between two references to the same variable SP). Low γ corresponded to good level of maintainability.

Most metrics have not been empirically validated.

3.8 Open Issues in Software Measurement

Software measurement issues surveyed in the previous sections mainly addresses software measurement for untimed transformational systems. A transformational system is a system which starts in an initial state for a given input and produces an output in its final terminating state. Modeling a transformation as a function between two successive states, a systematic construction of fundamental measures and measurement models were discussed.

A reactive system is not just transformational, for it has to have a certain level of synchronization. A real-time reactive system must have synchrony and timeliness. Thus, we need to have a rigorous and formal approach to real-time reactive software measurement.

There is insufficient work on the proper use of abstraction necessary for constructing measurement models, measurement of data collection, and analysis for real-time software. In particular, we need to address quality assessment, and the measures necessary to evaluate/predict quality of real-time reactive systems. These issues are addressed in the following chapters.

Further research is necessary to address system evolution, and the measures necessary to predict the reliability of evolving systems.

Chapter 4

Quality Model for Real-Time Reactive Systems

4.1 Notion of Quality

The concept "quality" is highly context-dependent. The quantitative characterization of a quality is referred to as a quality measure. The quality of a software product or a process can be characterized by the set of quality factors that, when evaluated, show if the software meets prescribed conditions. This may serve as a basis for reaching product release decisions, as an aid in interpreting whether improvements have been achieved, and for many other purposes.
The model of software quality aims to predict the level of quality of the software product, and is based on quality factors. The problem is to incorporate the right factors into the quality model, to define the relevant criteria for the quality factors in that model, and to develop measures that will be correctly characterizing those criteria. The following are some of the views of software quality and of the quality measurement [KP96]:

1. *Measuring the User's View*

   The users expect usability, reliability and performance of the software product.

2. *Measuring the Manufacturing View*

   The suggested characteristics to measure are the *defect counts* and *rework costs.*

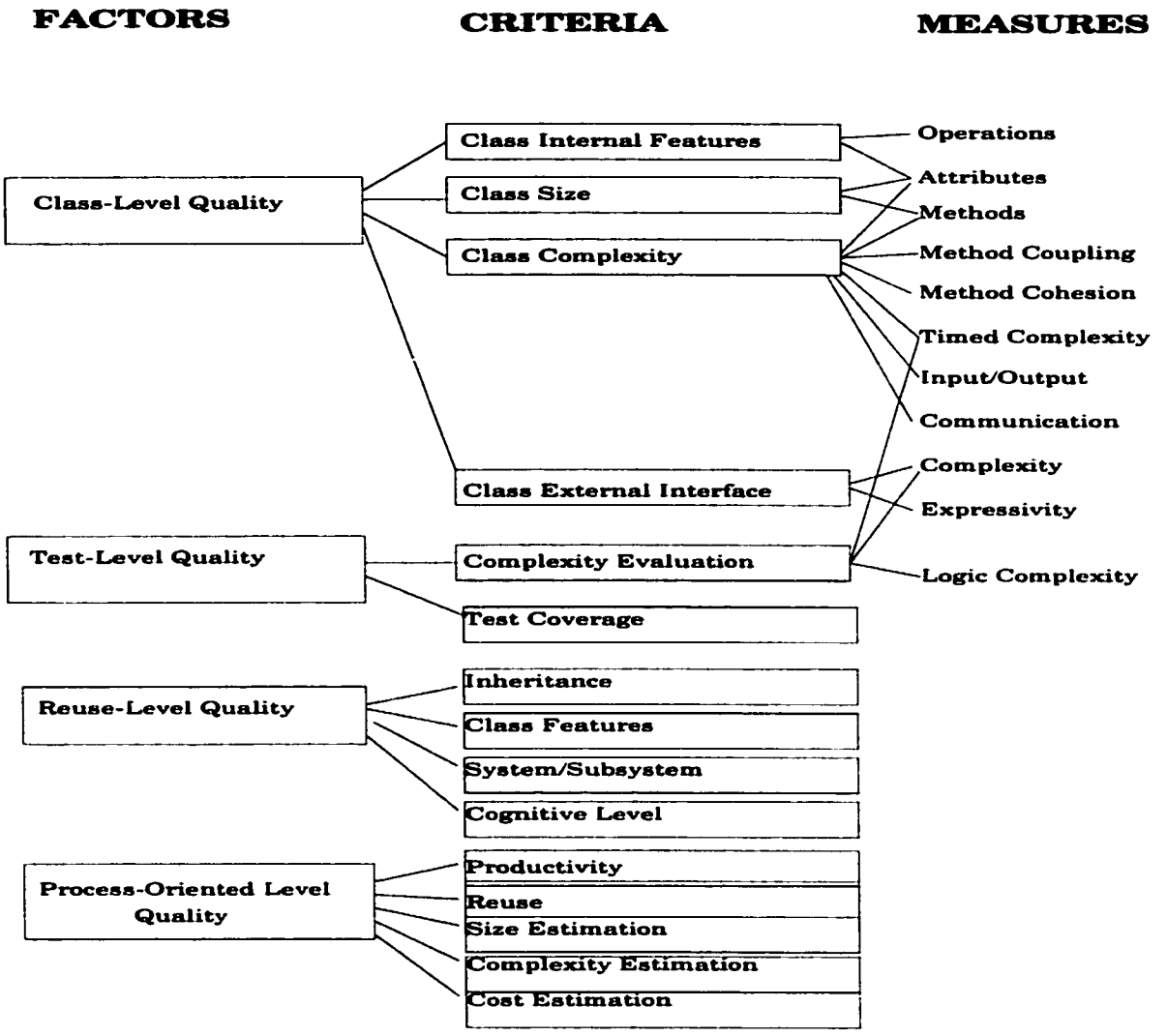


Figure 22: Nesi and Campanai's FCM Quality Model for Real-Time Software

3. *Measuring the Product View*

   A product view of quality considers the product's internal quality attributes. The measurement of internal quality indicators assess external quality. Validated models that link the product view to the user's view are needed to confirm that internal product quality assures external quality.

4. *Measuring the Value-Based View*

   This view is considering the trade-offs between cost and quality. The measurement is necessary to compare the product cost with the potential benefits.

4.2 Existing Quality Measurement Models

The aim of quality control is to provide quality of the final product that is satisfactory according to the quality requirements defined a priory.

The well-known existing quality models are the *McCall's quality model* [M77], *Boehm's quality model* [BBK78], *ISO 9126 Standard Quality Model* [ISO91], *Factor-Criteria-Metrics quality model* [IEEE93]. In [W97] the quality of object-oriented design is modeled in terms of static design characteristics of *size*, *complexity* , *coupling*, *sufficiency*, *completeness*, *cohesion*, *primitiveness*, *similarity*, and *volatility*. The characteristics have been defined in terms of the formal model of objects (based on categories), developed by the author to use as a basis for static measurement of a software design. The proposed measure construction technique [W97] has been applied to construct measures of the above characteristics. Each measure has been validated theoretically by showing how the measure satisfies the requirements of the quality characteristic.

Among the most widely accepted models for evaluation of the quality of the product that can be followed to ensure product quality, we mention ISO/IEC Standard ISO-9126 [ISO91], and IEEE Std 1061-1992 [IEEE93].

ISO 9126 provides the definition of the characteristics and associated quality evaluation process to be used when specifying the requirements for and evaluating the quality of software products throughout their life cycle. The characteristics are *functionality*, *reliability*, *usability*, *efficiency*, *maintainability* and *portability*. This standard does not provide sub-characteristics and metrics, nor the method for measurement, rating and assessment.

In *Software Quality Factor-Criteria-Metrics Framework* [IEEE93] the technique to build a quality model according to the organization goals and management requirements, is defined. The technique to build such models is also called *Goal-Question-Metric* approach and is described in [FP97]. Quality models have hierarchical structure based on the decomposition of quality requirements into measurable components. The hierarchical model is useful because it provides flexibility when modification of the model is necessary. The model's flexible hierarchical structure is obtained by decomposition of every *quality requirement* in quality attributes (*factors*) from the management and user - oriented views; decomposition of each factor in software -

oriented attributes (*criteria*) from the technical personnel views. Quantitative representation (software metrics) of the characteristics of each criteria are identified and associated to the established criteria and factors.
Both approaches are important and both require the presence of a system for managing quality.
There is, however, a lot of discussion on these standards and the certificates accompanying them. Adhering to strict software–quality standards can in some circumstances be counterproductive because the software development process has to be attuned to the specific systems development environment. Fenton et al. [FP97] suggest the monitoring of the software quality in two different ways:

1. **Fixed model approach**. This approach assumes that the quality factors needed to monitor a project are a (sub)set of those in already published model.

2. **Define your own quality model approach**.

Real-time software is considered to be different from other software due to its time constraint feature. Applying measures, created for other approaches, to quantify the factors of real-time reactive systems quality would lead to invalid measurement. In the measurement of the real-time systems, a lack of adherence to theoretically valid principles is still present. In addition, real-time reactive systems measurement involves specific need to give more evidence to system's behavior, thus limiting the applicability of existing quality models and measures. Figure 22 shows a model for controlling the quality of real-time development process, as presented in [NC96]. The implementation of the measurement model has been integrated in the development process, mainly focused on the specification of embedded reactive systems. The above measurement framework has allowed flexibility in the estimation of system quality characteristics at each phase of software life-cycle using a characteristic's weight (depending on the phase) technique. A set of metrics has been selected on the basis of their intuitive correlation with the respective quality characteristics, and their efficiency has been confirmed by experimental data, based on experimental work. Acceptable ranges of measures have been established. Most of the measures have been newly defined, while the others have been defined as equivalent to existing object oriented measures. The measurement framework is believed to be useful as objective support for developers in all phases of software life-cycle, even if the system is only

partially specified, and to be applicable to other object-oriented approaches with minor changes. The week point in the above work is the lack of theoretical validation of the proposed measures. Consequently, there is no guarantee of correctness of the quality model when applied in a different environment.

4.3 Quality Model

Our approach is to develop a model for controlling the quality in real-time reactive systems based on the most widely accepted characteristics of software ([IEEE93], [ISO91], [FP97], [BaPS98], [BBM96], [BG93], [Bi96], [CK94], [Ch97], [COTT95], [EM97]), and the previous work carried out in the TROMLAB. The quality is decomposed from the management and user-oriented views in quality factors **complexity**, **maintainability**, **testability**, **test adequacy**, **reliability** and **functionality**. Figure 23 illustrates the relationship between object oriented quality factors, criteria and metrics. In the following chapters we develop a definition of each one of these factors as well as the decomposition of each factor in software - oriented attributes (criteria). We have chosen the *define your own quality model* approach to develop a measurement framework for the quality control in the TROMLAB environment for the following two reasons:

- Existing models and measures do not apply to real-time reactive systems. Measurement of a real-time reactive system should not only include its structural information but also its dynamic information. For instance, a controller object that interacts with three trains and one gate is structurally and behaviorally different from a controller object that interacts with five trains and two controllers. Yet, these two controller objects in TROMLAB can be initialized from the same generic reactive class.

- Most of real-time reactive systems operate in safety-critical applications.

For instance, a safety property of the *Train-Gate-Controller* system is that

> **whenever a train is in the crossing the controller should be monitoring the gate and the gate must remain closed**

This safety property should be verified in the system design and must be enforced in an implementation of the design. Hence, quality measurement can not be restricted to the code level. It must be applied at the specification, design, and implementation levels.

The quality factors have been defined in terms of the TROM formal model, and are to be used as a basis for quality assessment of a software development in the TROMLAB framework. In order to guarantee the validness of the measures, each measure has been constructed from theory out, and the representation tested for tracking and consistency.

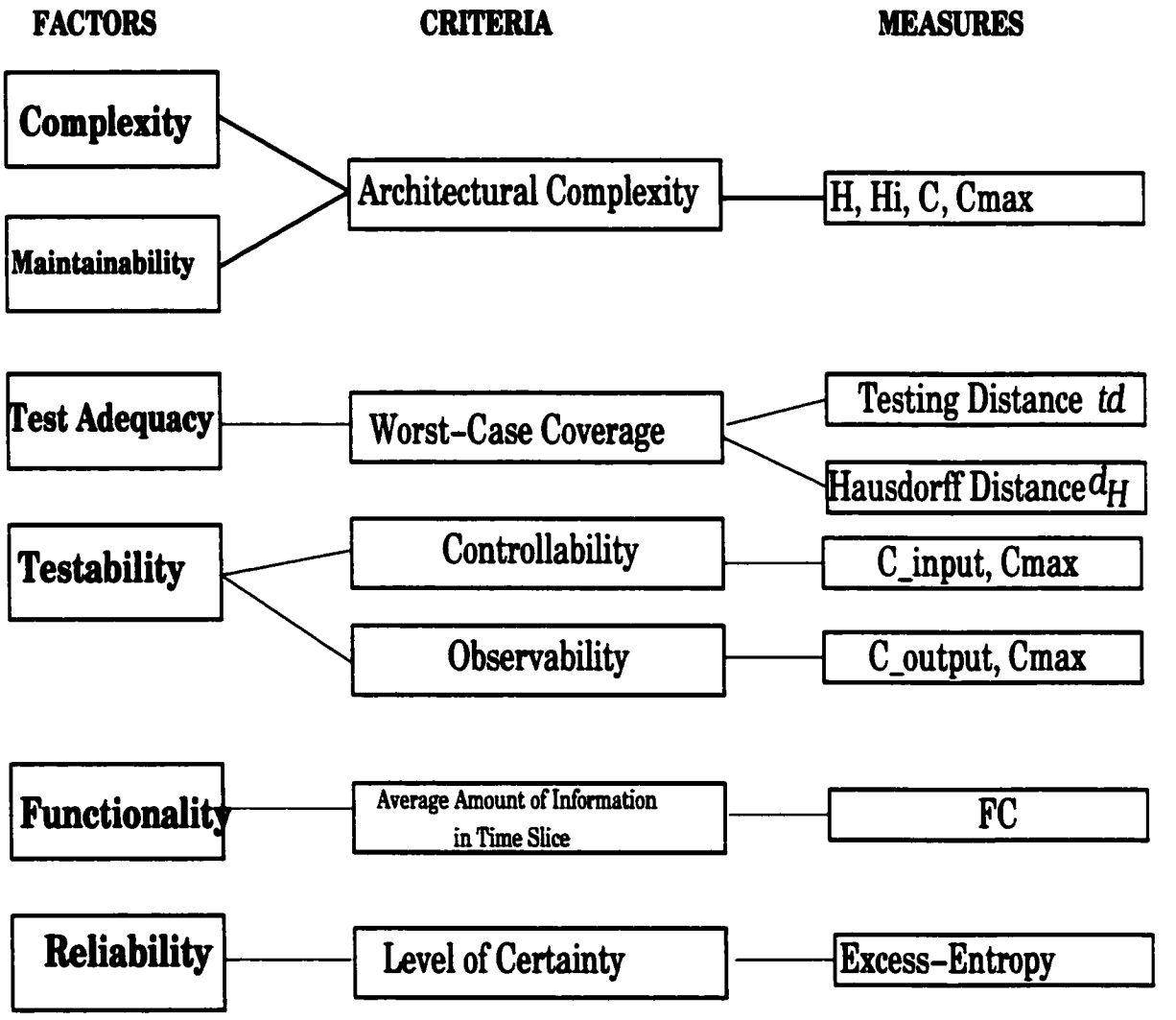


Figure 23: *FCM Quality Model for Real-Time Reactive Systems*

Chapter 5

Measurement related to Complexity

5.1 Notion of Complexity

The term software complexity, referring to the difficulty in comprehension, development and analysis, was introduced in late 1960. Complexity is an internal quality attribute that indicates the degree to which a system/component has a design/implementation which is difficult to understand and verify. There are different interpretations of complexity from the measurement perspective, namely: **algorithmic** complexity of the algorithm implemented to solve the problem; **computational** complexity of the underlying problem; **cognitive** complexity (i.e., difficulty of understanding a problem); and **structural** complexity. In the present work the perspective is to measure the structural complexity of the high-level software description and software design, thus when referring to complexity we will mean the *structural complexity* of the software. The structural complexity is decomposed into **syntax complexity, inheritance complexity, and interaction complexity** [KKKC96].

Complexity is usually the main factor on which the quality control of the object oriented development process depends. In this chapter, we review the existing complexity measures, and propose measurement to quantify the level of complexity in real-time reactive systems development.

There is not much work done in the area of complexity measurement and management for real-time reactive systems. Our approaches to assess architectural complexity,

testing complexity, and implementation complexity are new. A static analysis technique useful in design understanding - *architectural slicing*, is introduced and used in our maintainability measurement model. We discuss the different complexities in the context of the process model shown in Figure 1 for developing real-time reactive systems, propose measurement to quantify the level of complexity in different phases, and methods to manage them.
Much of our work seems applicable to frameworks and systems other than TROMLAB as long as it is modeled on labeled transition systems in an object-oriented paradigm.

5.2 Architectural Complexity Measurement

The total complexity of an object-oriented system's design is composed from the *internal complexity* (of the classes) and the *interface/architectural complexity* (a function of the relationships among the components of the software).
Major factors of software complexity are cohesion, coupling, modularity, and module complexity. There are a few existing sets of object-oriented design software metrics specifically designed to measure the *static complexity in the design of classes.*
In the following sections a number of existing object-oriented design static complexity metrics are described.

5.2.1 Existing Measures

We will focus on metrics that allow the prediction of future development efforts, are objective (do not depend on the judgment of a human user and can be preferably expressed in a machine-executable algorithm), and can be applied at reasonable cost. In this chapter, the above mentioned metrics are described in detail, and are summarized in the Factor - Criteria - Metrics quality model (Figure 24).

MOOSE set of OO Metrics

The Chidamber and Kemerer's (MOOSE) [CK94] set of six implementation-independent metrics *WMC, DIT, MOC, CBO, RFC, LCOM* are specifically designed to measure the *complexity in the design of classes*, as it is the central issue in the object oriented paradigm. The metrics are based on measurement theory and are evaluated analytically.

Weighted Methods Per Class (WMC)

Considering a Class C with methods $M_1, \ldots M_n$ defined in the class, let c_i be the static complexity of the method M_i.

Definition:

$WMC = \sum_{i=1}^{n} c_i$

Range of metric's values: $[0 \ldots N_{WMC}]$.

A higher value indicates higher static complexity of the class.

Depth of Inheritance Tree (DIT)

Definition:

DIT = *Maximum Length from the Node to the Root of Inheritance Tree*

Range of metric's values: $[0 \ldots N_{DIT}]$.

A higher value indicates higher complexity of the design and class maintenance effort.

Number of Children (NOC)

Definition:

NOC = *Number of Immediate Subclasses Subordinated to a Class in the Class Hierarchy*

Range of metric's values: $[0 \ldots N_{NOC}]$.

A higher value indicates higher complexity of the design and class maintenance effort.

Coupling Between Objects/Classes (CBO)

Definition:

$CBO(Class)$ = *Number of Other Classes to which the Class is Coupled*

Range of metric's values: $[0 \ldots N_{CBO}]$.

A higher value indicates higher complexity of the design.

Response For a Class (RFC)

Definition:

RFC = *Number of Methods that can be Invoked in Response to a Message to an Object of the Class*

Range of metric's values: $[0 \ldots N_{RFC}]$.

A higher value indicates higher class maintenance effort.

Lack of Cohesion in Methods (LCOM)

Let $P = \{$ *Method Pairs Whose Similarity is 0* $\}$,
and $Q = \{$ *Method Pairs Whose Similarity is Not 0* $\}$

Definition:

$$LCOM = \begin{cases} |P| - |Q|, & if \quad |P| > |Q| \\ 0, & otherwise \end{cases}$$

Range of metric's values: $[0 \ldots N_{LCOM}]$.
A higher value indicates higher class maintenance effort.
The six metrics are designed to measure the three non-implementation steps in Booch's OOD methodology [Chidamber 1994] :

1. Identification of Classes (Objects) *(WMC, DIT, NOC)*
2. Identification of the Semantics of Classes (Objects) *(WMC, RFC, LCOM)*
3. Identification of the Relationships Between Classes (Objects) *(RFC, CBO)*

The metrics are based on the measurement theory and are evaluated analytically against the subset of Weyuker's axioms of complexity in [CK94].

In [BBM96] the MOOSE metrics *RFC, NOC, DIT, CBO, WMC* are **empirically validated as predictors of fault-prone classes** if used early in the life-cycle (high- or low-level design).
Five of the MOOSE metrics and other metrics defined in [LH93] have been empirically validated and reported to be adequate in *predicting the class maintainability.*

Li and Henry's Metrics

The metrics *MPC, DAC, NMC* defined in [LH93] have been empirically validated and reported to be adequate in *predicting the class maintainability.*

Message-Passing Coupling MPC

Definition:
MPC = *Number of Send Statements Defined in a Class*
Range of metric's values: $[0 \ldots N_{MPC}]$.
Higher value indicates higher level of dependency of the implementation of the local methods on the methods in other classes and higher maintenance effort.

Data Abstraction Coupling DAC

Definition:
The measure of the coupling complexity due to declaration of variables of Abstract Data Type within the class.
DAC = *Number of Abstract Data Types Defined in a Class*
Range of metric's values: $[0 \ldots N_{DAC}]$.
Higher value indicates higher level of coupling complexity and higher maintenance effort.

Number of Methods in a Class NMC

Definition:
Class interface increment metric.
NMC = *Number of Local Methods in a Class*
Range of metric's values: $[0 \ldots N_{NMC}]$.
Higher value indicates more complex class' interface and higher maintenance effort.

Defect Density and Rework Measurement: MOOD

The MOOD metrics [BaPS98] have been created to measure independent aspects of the design objectively, size independently and language independently. The MOOD metrics measure *encapsulation* (MHF and AHF), *inheritance* (MIF and AIF), *polymorphism* (POF), and *message-passing* (COF).

Method Hiding MHF

Let TC be a total number of classes; $M_d(C_i)$ - a total of methods defined (not inherited); $V(M_{mi})$ - the measure of *visibility* (% of the total classes from which the

method M_{mi} can be called).
Definition:

$$MHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{M_d(C_i)} (1 - V(M_{mi}))}{\sum_{i=1}^{TC} M_d(C_i)}$$

Attribute Hiding AHF

Let $A_d(C_i)$ be a total of attributes defined (not inherited); $V(A_{mi})$ - a measure of *visibility* (% of total classes from which A_{mi} can be referenced).
Definition:

$$AHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{A_d(C_i)} (1 - V(A_{mi}))}{\sum_{i=1}^{TC} A_d(C_i)}$$

Method Inheritance MIF

Definition:

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

where
$M_a(C_i) = M_d(C_i) + M_i(C_i)$,
$M_a(C_i)$ is a total of available methods;
$M_d(C_i)$ is a total of methods defined;
$M_i(C_i)$ is a total of inherited methods.

Attribute Inheritance AIF

Definition:

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum i = 1^{TC} A_a(C_i)}$$

where
$A_a(C_i) = A_d(C_i) + A_i(C_i)$,
$A_a(C_i)$ is a total of available attributes,
$A_d(C_i)$ is a total of attributes defined,
$A_i(C_i)$ is a total of attributes inherited.

Polymorphism *POF*

Definition:

$$POF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC}[M_n(C_i) * DC(C_i)]}$$

where
$M_d(C_i) = M_n(C_i) + M_o(C_i)$,
$DC(C_i)$ is the number of descendants; $M_n(C_i)$ is a number of new methods
$M_o(C_i)$ is a number of overriding/redefined methods.

Coupling *COF*

Definition:

$$COF = \frac{\sum_{i+1}^{TC}[\sum_{j=1}^{TC} is - client(C_i, C_j)]}{TC^2 - TC}$$

where $(TC^2 - TC)$ is the maximum number of couplings in a system with TC classes.

$$is - client(C_c, C_s) = \{ \begin{matrix} 1. & iff\ the\ client - server\ relation\ exists \\ 0. & otherwise \end{matrix}$$

Note: the client-server relation exists if the client class contains at least one non-inheritance reference to a(method or attribute of a supplier class C_s.

The measurement's effect on quality has been assessed individually for each MOOD's metric by determining the correlation between the MOOD design metrics and the object oriented implementation quality measures of *defect (fault and error) density* and *normalized rework (effort spent on repairing errors)*. MIF, COF, MHF and POF were empirically validated proving to be statistically significant for **predicting the implementation quality criteria** *defect density* **(reliability factor) and** *rework* **(maintainability factor)** during the design phase.

5.2.2 Related Work

In order to be able to manage the complexity of the design, the information on the actual level of complexity is needed.

Real-time systems are considered to be different from other software due to their time constraints. A recent study [CRS96] analyzes the applicability of traditional software complexity measures to real-time software complexity measurement. The results of the study show that the information flow measures seem to be more suitable for real-time software than the classic control flow complexity measures.
In this section we propose new measures to quantify the level of complexity of the reactive systems in the TROMLAB environment. We accept the results of the above analysis and extend the choice of information theory based measurement for the design complexity at architectural level. We separate information flow measures into in-flowing and out-flowing measures for the testability, i.e., testing complexity prediction, measurement purposes. We assess the complexity in the maintenance phase based on a new technique named *architectural slicing* visualizing the architectural complexity of system's components, and their interconnections.

5.2.3 Approach

In TROMLAB, the design complexity is concerned with the behavior of the reactive objects and the level of interaction between the objects. Let us consider an application A and a system S that solves A. The system S can be constructed as a collection of reactive objects $\{O_1, \ldots O_n\}$ that interact among themselves in solving the problem A. Each object's *state* of S is described by the set of static attributes and their current values. An object's behavior is concerned with object's state changes (internal behavior) and message passing (request from another object to perform a service) (external behavior). Therefore, the total structural complexity of the software system is a function of the *internal complexity* (of the object's internal behavior) and the *architectural complexity* (of the interactions between the objects). The internal complexity is mainly an implementation issue, and any structural complexity metric would be applicable (for example, McCabe's cyclomatic complexity [Mc76]). At the design level, we are concerned with the architectural complexity only.
The architectural complexity may arise in two ways: the presence of a large number of components and the fact that most of these influence many others. In our work, the architectural complexity is viewed in terms of how the software components interact through message passing mechanism, without considering the complexity of its

components. To manage complexity, it is sometimes possible to decompose hierarchically the system into a few subsystems with relatively weak interactions between them. In their turn, each of this subsystems may be subjected to the same treatment. The hierarchical decomposition of complexity then corresponds to the decomposition of the total amount of information transfer. The information transfer between the components is synonymous with the complexity of interactions within the software system, so measuring the amount of information transfer is assessing the complexity management in real-time reactive systems.

5.2.4 Requirements for Complexity Measures

The purpose of the architectural complexity measurement is a comparison of different designs. Therefore the chosen scale is the ordinal one, and the constraint of the chosen scale type for the ordinal representations (*Cantor's Theorem*) is: $\forall a, b \in A : a \preceq b \iff \phi(a) \leq \phi(b)$

The sufficient condition for the representation ϕ to exist is:

A *is finite set* and $\preceq$ *is a weak order* (*connectedness, transitivity*)

In our work, the mathematical model underlying the notion of coupling is a graph whose nodes represent design objects and edges represent the connections between them (see Figure 25). Let us denote a graph by $G = \langle V, E \rangle$, where V is the set of design objects and E is the set of communication links in V (we denote by aRb the link R between the objects $a, b \in V$). Here, a design component means an entire graph or any connected component in a graph.

The following properties should be met by all measures based on coupling:

1. *(coupling is nonnegative)*

   $coupling(G) \geq 0$

2. *(coupling can be null)*

   $\forall a \in V : (\{e \in E, b \in V | e = aRb\} = \emptyset\}) \Rightarrow coupling(G) = 0$

3. *(adding an inter component relationship does not decrease coupling)*

   $\forall G_1 = \langle V_1, E_1 \rangle, G_2 = \langle V_2, E_2 \rangle :$

   $(\exists e \in E_2 \bullet e \notin E_1 \wedge (E_2 = E_1 \cup e)) \rightarrow coupling(G_2) \geq coupling(G_1)$

4. *(merging ($\circ$) two unconnected components does not change coupling)*

   $\forall G_1 = \langle V_1, E_1 \rangle, G_2 = \langle V_2, E_2 \rangle :$

   $(E_1 \cap E_2 = \emptyset \wedge G_3 = G_1 \cup G_2) \Rightarrow coupling(G_1 \circ G_2) = coupling(G_3)$

5. *Coupling forms a weak order; design components can be ordered in terms of level of coupling they contain*

The mapping between the empirical structure and the numerical structure is described by the mathematical model of the measure.

5.2.5 Mathematical Model

The architectural description in **TROMLAB** has two aspects to it:

static aspect: class structure diagrams, refinements of classes, and state machine descriptions.

dynamic aspect: system configuration describes the objects in a subsystem and their interactions.

Objects interact through message passing and synchronously change their states during an interaction. For instance, when an object o_1 sends the message $e!$ to object o_2, the message $e?$ occurs for object o_2, and the objects change their states simultaneously.

For the architectural complexity measurement purposes, all the information related to the interaction between objects via message exchanges, has to be extracted from the TROMLAB design specification. The architectural complexity measures have to quantify objectively the amount of information exchanged between the objects. Information theory based software measurement is used to quantify the interaction complexity in terms of an amount of information, based on some abstraction of that interaction. Measurement based on information theory is objective because it evaluates the abstraction of a software product independently of the software notation to be used and the life cycle phase to be applied on. We define an *object-predicate* table, in which each row is an object in the system and each column is a port in the system. This table abstracts the interaction between objects:

$$Object\_Predicate\_Table(object_i, port_j) = \begin{cases} 1, & \text{if port}_j \text{ belongs to object}_i \\ & \text{or is linked to a port of object}_i, \\ 0, & \text{if this is not the case.} \end{cases}$$

We quantify the complexity of interaction in terms of *excess-entropy* on the object-predicate table abstraction. In the past Emden's original work on quantification of information transfer model [E70] was adapted for calculating the level of complexity of code [RB89], and for measuring the maintainability level of software specifications [COTT95]. Ours is the first attempt to adapt the information transfer model to reactive systems.

The excess-entropy C is defined as the difference between the sum of the entropies of parts and the entropy of the whole. Let us consider an m-partition of a set of n components such that $\sum_{i=1}^{m} n_i = n$. The set of non-negative numbers $\{\frac{n_1}{n}, \ldots, \frac{n_m}{n}\}$ is associated to the m partitions and $\sum_{i=1}^{m} \frac{n_i}{n} = 1$. The formulas for calculating the entropy H and the excess-entropy C are:

(A): $H = \log_2 n \; - \; \frac{1}{n} \sum_{i=1}^{m} n_i \log_2 n_i$

(B): $C = \sum_{i=1}^{m} H_i \; - \; H$

(C): $H_i = \log_2 n_i \; - \; \frac{1}{n_i} \sum_{j=1}^{k_i} p_j \log_2 p_j$

where H_i is the entropy of a partition$_i$, k_i is the number of different column configurations in the object-oriented table corresponding to the partition$_i$, p_j is the number of columns with the same configuration$_j$, and $n_i = \sum_{j=1}^{k_i} p_j$. where H_i is the entropy of a partition$_i$.

Justification for choosing a row configuration as criteria for partition. The row configuration indicates the channels involved in the communication process of the corresponding object. Saying that row configurations are identical means that both rows are communicating the same objects (RE: partition to which they belong to) and they form a (sub)partition.

The calculation of H_i depends on the interconnections between the objects that belong to the partition$_i$. If it is possible to decompose a partition into subsets with relatively weak interactions between them, then H_i is calculated using the entropy formula **(A)**. Otherwise formula **(C)** is used, where k_i is the number of different row configurations in the object-oriented table corresponding to the partition$_i$, p_j is the number of rows with the same configuration$_j$, and $n_i = \sum_{j=1}^{k_i} p_j$.

In **TROMLAB**, the port links effectively determine the set of all valid messages that can be exchanged among the objects through their ports. Therefore, in order to define the abstraction of interactions as an object-predicate table, we consider the port names as predicates that associate one object with another (through communication channels). The reliability of the software measurement depends on the rigor of in the measurement approach. We follow the representational approach that requires the definition of an abstract reactive model, formal description of the empirical view of complexity in terms of that model, and the theoretically valid representation (measure) for the complexity.

5.2.6 Architectural Complexity Measures

To quantify the architectural complexity in TROMLAB environment, let us consider an m-partition of the set of n ports in mutually independent subsets such that n_i ports of each subset$_i$ (i=1,m) are connected by means of message passing through port links. Clearly each number $\frac{n_i}{n}$ indicates the probability that a port belongs to the partition$_i$. The entropy H_i is calculated considering only n_i ports in the subset$_i$. In order to measure the level of interconnection independently of the size of the system description, the following *Architectural Complexity* measure is proposed:

$$AC = \frac{C}{C_{max}}$$

$$C_{max} = (n - 1)\ log_2\ n$$

Justification for C_{max}. In case all the n elements in a partition are interconnected, then $m = n$, $H = log_2\ n$; $\forall i \in \{1, \ldots n\} : \ H_i = log_2\ n$. The quantity C_{max} represents the maximum excess-entropy of a set of components and thus logically normalizes the measure.

The range of the values of the AC measure is $[0, 1]$. The value 0 indicates that there is no common information between the elements of a set, and the value 1 indicates that all ports are linked.

To locally evaluate the contribution of a port to the architectural complexity of a design, the *Local Architectural Complexity (LAC)* measure which reflects the relationship between the individual interconnectivity level of a port$_i$ (C_i) and the level of global interconnectivity (C_{max}) of the set.

The formula for evaluating LAC depends on the cases considered in the previous section, namely:

1. *The given object belongs to only one set of interacting objects.*

   LAC is defined as a ratio between the weight of the interconnections for a given object (H_i) and the maximum interconnectivity (C_{max}) of the architecture:

   $LAC_i = \frac{H_i}{C_{max}}$

2. *The given object belongs to at least two sets of interacting objects.*

   LAC is defined as the ratio between the sum of the weights of all the sets j of interacting objects the given object belongs to, and the maximum interconnectivity (C_{max}) of the architecture:

   $LAC = \frac{\sum_j H_j}{C_{max}}$

The LAC measure values near 0 indicate that the influence of the object on the global maintainability of the system is small, i.e., it is easy to maintain, and the bigger value of LAC indicates more interactions between the objects.

We illustrate the architectural complexity measure computation on the *Train - Gate - Controller* system shown in the Figure 18. This system is configured with 5 trains (objects t_1, t_2, t_3, t_4, t_5), 2 controllers (objects c_1, c_2) and two gates (objects g_1, g_2). In this configuration, the object *train3* is allowed to interact with both controllers, while the other train objects can only interact with one of the controllers. This schematic drawing conforms to a system with different trains on specific routes. The interactions are represented as a graph in Figure 25. The object-predicate tables for the two connected components of the graph 25 are given in Figure 26.
There are two subsets resulting from the partition of the initial set of sixteen ports. The first subset is composed of $\{C1, C2, C3, P1, P2, P3, G1, S1\}$, and the second one of $\{C4, C5, C6, P4, P5, P6, G2, S2\}$. The values of the variables, in the Emden's model, for this case are: $n = 16$, $m = 2$, $n_1 = n_2 = 8$, $k_1 = k_2 = 4$, $\forall\, j = 1,4 \bullet p_j = 2$. The following expression gives the value for AC:

$$AC = \frac{2(\log_2 8 - \frac{1}{8}\sum_{i=1}^{4} 2\log_2 2) - (\log_2 16 - \frac{1}{16}\sum_{i=1}^{2} 8\log_2 8)}{15\log_2 16}$$

- $n = 16$, $m = 2$, $n_1 = n_2 = 8$, $k_1 = k_2 = 4$, $\forall\, j \in \{1 \ldots 4\} : p_j = 2$
- $\forall i \in \{1, 2\} : \quad H_i = \log_2 8 - \frac{1}{8} \sum_{i=1}^{4} 2 \log_2 2$
- $H = \log_2 16 - \frac{1}{16} \sum_{i=1}^{2} 8 \log_2 8$
- $C_{max} = 15 \log_2 16$
- LAC for the objects t_1, t_2, c_1, g_1: $LAC = \frac{H_1}{C_{max}} = 0.03$
- LAC for the objects t_4, t_5, c_2, g_2: $LAC = \frac{H_2}{C_{max}} = 0.03$
- LAC for the object T_3: $LAC = \frac{H_1}{C_{max}} + \frac{H_2}{C_{max}} = 0.06$

Testing AC and LAC

The second version of the *5 trains - 2 gates - 2 controllers* subsystem configuration shown in Figure 27 is more complex due to the added connection between the objects t_1 and c_2 (i.e., $version_1 \preceq version_2$). The added connection between the objects t_1 and c_2 affects the values for the following model variables (compared to version 1): $n = 18$, $n_2 = 10$, $k_2 = 5$, $H_2 = \log_2 18 - \frac{1}{18} \sum_{i=1}^{5} 2 \log_2 2$, $C_{max} = 17 \log_2 18$.
The above testing of the AC and LAC models with real data has demonstrated their tracking and consistency with changes in the architecture.

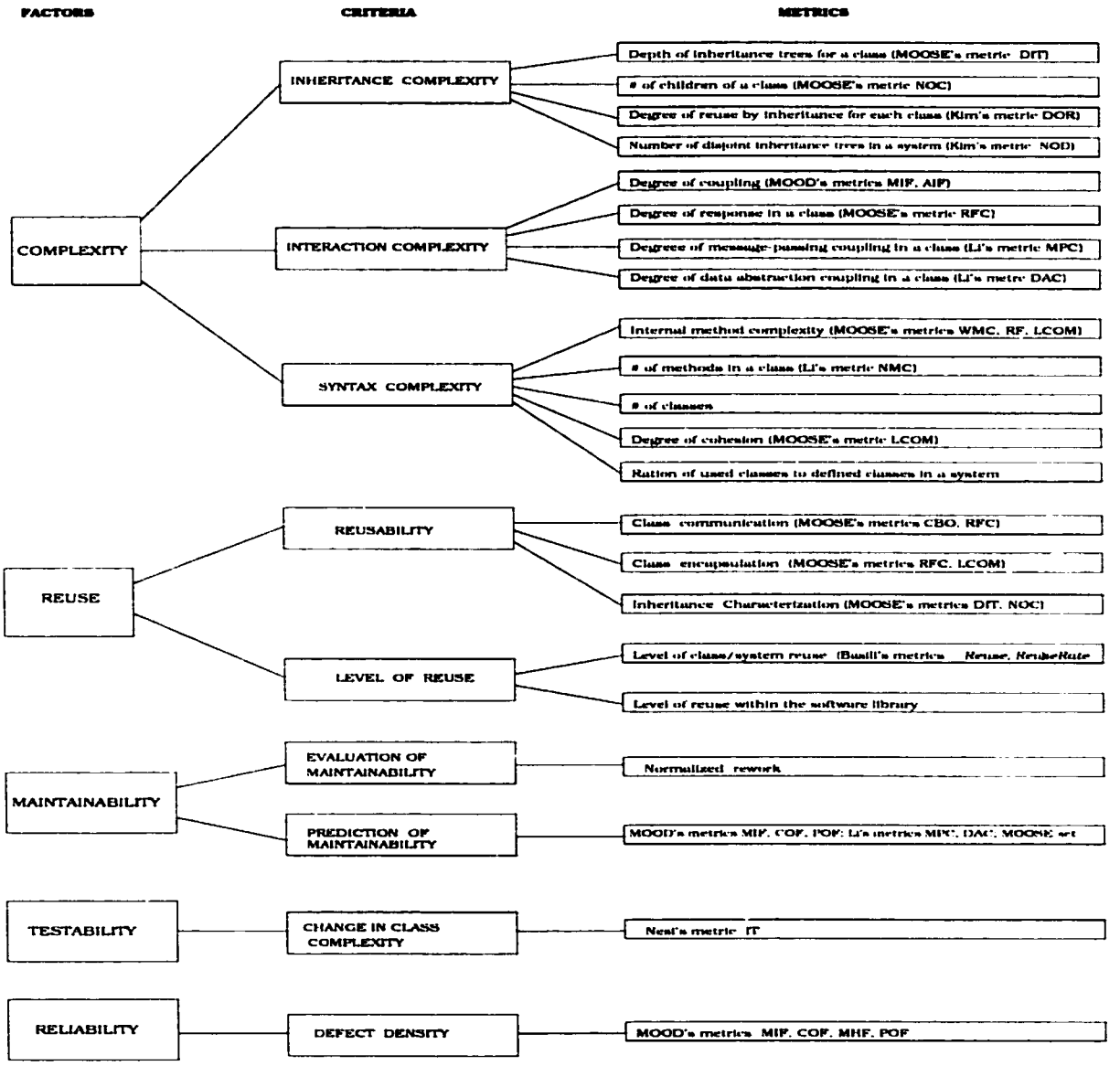


Figure 24: FCM model of OOD Quality (based on *MOOSE* set, *MOOD* set and *Li and Henry*'s Metrics).

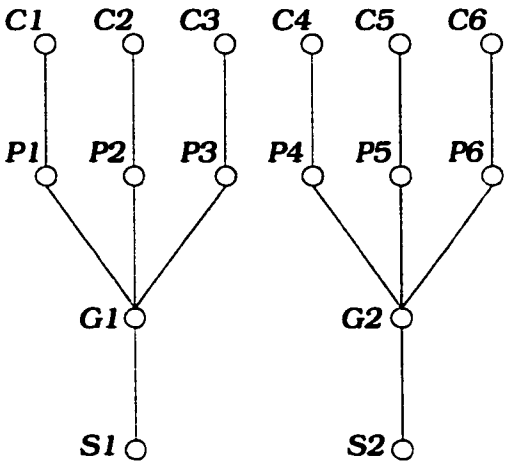


Figure 25: Train-Gate-Controller2 Subsystem: graph for communication links

	t1	t2	t3	c1	g1
C1	1			1	
C2		1		1	
C3			1	1	
P1	1			1	
P2		1		1	
P3			1	1	
G1				1	1
S1				1	1

	t3	t4	t5	c2	g2
C4	1			1	
C5		1		1	
C6			1	1	
P4	1			1	
P5		1		1	
P6			1	1	
G2				1	1
S2				1	1

Figure 26: Object-Predicate Table abstraction for the connected components of the graph in Figure 25

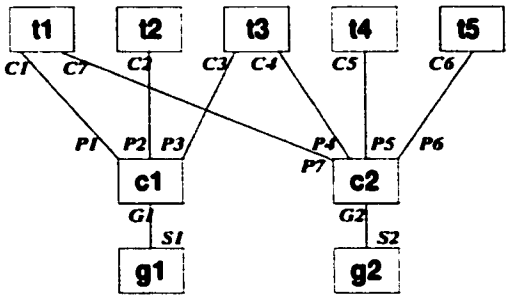


Figure 27: Collaboration Diagram for Train-Gate-Controller2 Subsystem: Version 2

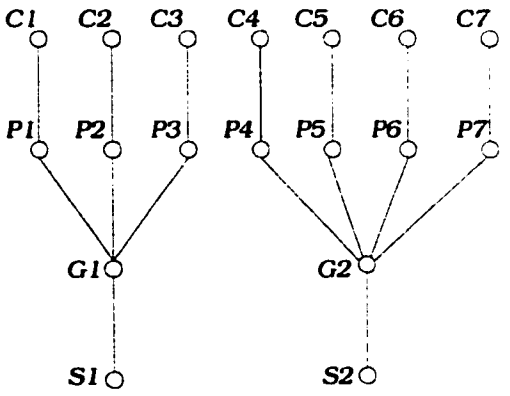


Figure 28: Version 2: graph for communication links for the connected components of the graph in Figure 27

5.3 Maintainability Measurement

The goal of this section is to assess the maintenance prediction based on the complexity of the designs, and to propose methods to localize changes in the designs during the maintenance phase.

Software maintenance is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment. Software maintenance is classified into four types: corrective, adaptive, perfective and preventive. Corrective maintenance refers to fixing a program. Adaptive maintenance refers to modifications that adapt to changes in the data environment, such as new product codes or new file organization or changes in the hardware of software environments. Perfective maintenance refers to enhancements: making the product better, faster, smaller, better documented, cleaner structured, with more functions or reports. The preventive maintenance is defined as the work that is done in order to try to prevent malfunctions or improve maintainability.

The system maintenance effort is evaluated based on two criteria, namely, the normalized rework and duration of the rework. The *normalized rework* is the average effort spent in realizing maintenance activities (i.e., *corrective, adaptive, preventive or perfective*), and has to be evaluated during the maintenance phase. The *duration of the rework* is the speed of implementing the change. The related process metric proposed in [FP97] is the *Mean Time to Repair (MTTR)*.

We usually think of software maintenance as beginning when the completed product is delivered to the client. While this is formally true, in fact decisions that affect the maintenance of the product are made from the earliest stage of design. A critical aspect of any object-oriented system design is its software architecture that describes a high-level organization as an interacting set of computational elements. Preserving and/or increasing the reliability of software during maintenance requires that software engineers understand how various components of a design interact. When a software system is not designed for maintenance, it exhibits a lack of stability under change. A modification in one part of the system has side effects that ripple throughout the system, turning "one small change" into a major rewrite of the product. Thus, the main challenges in software maintenance are to understand existing software and to make changes without introducing new bugs.

Maintainability is a prediction of the ease with which a system can evolve from its current state to its future desired state. Maintainability indicates the maintenance effort in the object-oriented system, and depends on the level of objects coupling through communication mechanisms provided by the object-oriented paradigm. The ever-changing world makes maintainability a strong quality requirement for the majority of software systems.
The maintainability measurement during the development phases estimates the maintenance effort in the object-oriented system, and therefore evaluates the likelihood that the software product will be easy to maintain. Maintainability measurement can be useful in comparing alternative designs in terms of their maintainability.

5.3.1 Related Work

The maintainability cost model is reflecting the decomposition of the factor maintainability in quality criteria. IEEE Standard Std 1061-1992 [IEEE93] suggests the following **Factor-Criteria-Metric** model for the Maintainability Factor:

Correctability Criteria - The degree of effort required to correct errors in software and cope with user complaints. The metric types suggested are *Effort* and *Fault Counts.*

Expandability Criteria - The degree of effort required to improve or modify the efficiency of functions of software. The metric type suggested are *Degree of Testing* and *Effort.*

Testability Criteria - The effort required to test software. The metric types suggested are *Effort* and *Change Counts.*

The idea to use software complexity measures in order to predict the maintenance effort and/or to compare designs in terms of their maintainability, is not new. Various object-oriented software design complexity measures have been published as a mean to ascertain the maintainability of the implementation.
In [WKHZKRSP2001], some enhancements to existing object-oriented code measures, and additional code measures are proposed, based on experience with measuring C++ and Java codes.

In [LV2000], the quantitative assessment of the impact of requirements changes on maintenance phase, and the quantitative estimation of costs of those changes, are addressed. The above approach involves product measurement to characterize quantitatively the elements in the process and the product models.
In [Sch99], software product and process measures are used to predict the stability of a software maintenance process in terms of reliability and risk of deploying the software. The approach is illustrated on the NASA Space Shuttle flight software.

5.3.2 Approach

Maintenance can be difficult because of bad architectural design decisions. In our work, the maintainability is estimated based on the complexity of interactions between the objects in the design architecture. The architectural complexity is viewed in terms of how the software components interact through message passing mechanism (that is, a type of coupling), without considering the complexity of its components. Coupling predicts the difficulty of changing the program and the likelihood of errors introduced by these changes.
Preserving and/or increasing the reliability of software during maintenance requires that software engineers understand how various components of a design interact. We differentiate between a given modification (*primary modification*) and the changes resulting from it in the affected part of the system (*secondary modifications*).
Our approach consists in assessing quantitatively the maintenance effort related to the *secondary modifications* that are due to interactions between the objects. It visualizes the maintainability of the design objects (*maintainability profile*), and reduces the maintenance effort through *architectural slices* that would localize the set of design objects to be affected by a given change.
Our *maintainability profile* is a visualization of the maintainability level for each object, based on the complexity of interactions between the objects in the corresponding architectural slice.

5.3.3 Maintainability Profile

We propose a static analysis technique useful in design understanding - *Architectural Slicing.* Architectural slicing extracts all the design objects relevant to the computations of a given one. Before a developer changes a particular object, architectural slicing can be used to decompose the design into two parts, one to be affected by the change, and another that will be unaffected by the change. Then, design retest will be required with respect to the affected part only. Architectural slices can be used to reduce the effort in examining software by allowing a maintainer to focus attention on the set of objects to be affected by the change only, one object at a time. Before changing the design of a particular object, the maintainer can use the architectural slice profile to partition into two groups - one consisting of the parts that will be affected by the design change and the other consisting of parts that will be unaffected by the design change. Only the group that is affected by the design change is required to be subjected to further analysis.

This approach would help in selecting a better design from among alternative design choices. Moreover, the stress points that usually lead to difficulties during system maintenance are visualized graphically through maintainability profiles. We illustrate our approach on the train-gate-controller case study.

Architectural Slice Extraction

The graph for communication links allows the extraction of an architectural slice for a specific object. The graph is made up of connected components (i.e., equivalence classes of vertices under the "reachable from" relation). Each vertex corresponds to a communication channel label, therefore there is one set of interacting objects corresponding to each component. For the purpose of defining the architectural slice for a given object, we have to consider the following cases:

1. *The given object belongs to only one set of interacting objects.* In this case, the above set of objects would define the architectural slice for the participating objects.

2. *The given object belongs to at least two sets of interacting objects.* In general, one object may belong to different sets of interacting objects. In this case,

the architectural slice is the union of the sets to which the interacting object belongs.

Illustration of architectural slicing for Figure 18.

The objects in the system depicted in Figure 18 are t_1, t_2, t_3, t_4, t_5, c_1, c_2, g_1, and g_2. The slices for the objects t_1 and t_2 consist of the objects $\{t_1, t_2, t_3, c_1, g_1\}$. The slice for the object t_3 consists of the objects $\{t_1, t_2, t_3, t_4, t_5, c_1, g_1, c_2, g_2\}$. The slices for the object t_4 and t_5 consist of the objects $\{t_3, t_4, c_2, g_2\}$.

Maintainability Profile

The maintainability profile depicts graphically the relative weight of the complexities of the architectural slices for the design objects. To locally evaluate the complexity of an architectural slice (independently of the size of the system), the *Local Architectural Complexity (LAC)* measure is applied.

Illustration of the Maintainability Profile

The profile created for the system depicted in Figure 18 is given in Figure 29.

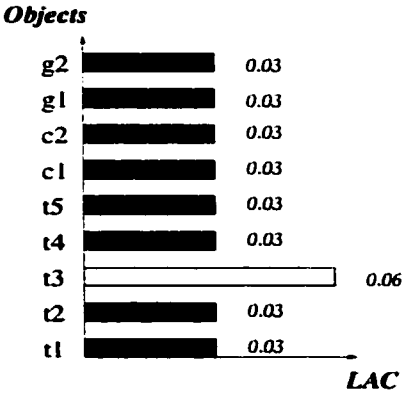


Figure 29: Maintainability Profile for Figure 18

We have tested the sensitivity of the maintainability profile to architectural complexity on two versions of the train-gate-controller model.
The second version of the train-gate-controller model is more complex due to the added connection between the objects t_1 and c_2 ($version_1 \preceq version_2$) (Figure 28). The architectural slicing, the maintainability profile and the analysis of the comparison of the two versions are presented below.

Architectural Slicing for Figure 27

The objects in the system depicted in Figure 27 are t_1, t_2, t_3, t_4, t_5, c_1, c_2, g_1, g_2.
The slices for the objects t_2, c_1, g_1 consist of the objects $\{t_1, t_2, t_3, c_1, g_1\}$.
The slice for the objects t_1 and t_3 consists of the objects $\{t_1, t_2, t_3, t_4, t_5, c_1, g_1, c_2, g_2\}$.
The slices for the object t_4 and t_5 consist of the objects $\{t_3, t_4, c_2, g_2\}$.
The values of LAC for the system objects are given in Figure 30.

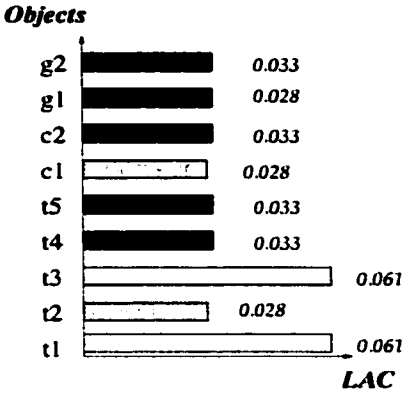


Figure 30: Maintainability profile for Figure 27

The comparison of the maintainability profiles for the two different configurations of the 5 Trains - 2 Controllers - 2 Gates systems shows higher relative complexity (value of LAC) for the objects t_1, g_2, c_2, t_4, t_5 of the more complex second case, thus illustrating the sensitivity of the LAC measure to changes in the complexity of the architecture, and ordering the design components in terms of the level of complexity they contain.

5.4 Testability Measurement

Software testing is a process of determining whether a software development has been correctly carried out. Software testing is necessary to produce highly reliable systems, since static verification techniques cannot detect all software faults. The testing process adds value to the software development process, thus the goals are to maximize the testing efficiency and to minimize the cost and difficulty of testing. The way to increase the testing efficiency is to efficiently allocate the testing resources according to the level of complexity of software components ([ZORS97]) and their level of testability, where the testability is the ability of the software to be easily tested [Bi96].

5.4.1 Notion of Testability

A software component's testability measurement is a prediction of software testing ability to detect faults. Binder argues that a software component with low level of testability would require more testing than a component with high level of testability. The potential benefits of measuring software testability are significant: testing resources can be distributed more effectively, and the degree to which software component's testing is to be performed could be estimated using the component's testability measure [VMM91].

5.4.2 Related Work

In [NC96] a testability estimation is proposed to identify the classes for which the test is needed before continuing with the development process. It is estimated relative to the last testing session, change in class complexity since last test:

$$TI_i = \frac{CC_i(ActualVersion) - CC_i(LastTestedVersion)}{CC_i(LastTestedVersion)}$$

where CC_i is the class complexity. Classes whose TI_i measure is higher than a predefined value must be tested before continuing with the development process.

Voas and Miller [VM93] propose static testability measurement approach, called Propagation-Infection-Execution. This method is based on input/output domains in random black-box testing. Le Traon and Robach [TR97] propose design specification

phase testability measurement in terms of controllability/observability of the software units that consider white box testing. The controllability and observability measures are based on information theory. Given software system's data flow specification, the measure predicts the controllability/observability of the software unit in terms of the information loss through the possible information flows.

Several design-for-testability strategies in terms of *controllability of the component's input* and *observability of its output* are suggested in [Bi96]. These are: *information hiding and separation of concerns; test scaffolding; adding explicit built-in test functions to report internal state on demand; defining a data type that exactly corresponds to the domain of the output variables.* To assess quantitatively the software testability Binder [Bi96] suggests the measure *Index of Testability* defined as the number of changes that must be made to obtain perfect observability and controllability of the software component. It is argued that software testability depends primarily on the characteristics of the software analysis and design descriptions (Factor *Representation*), characteristics of the implementation (Factor *Implementation*), built-in test capabilities that provides built-in observability and effective control of the software (Factor *Built-In Test*), and overall software process maturity that views testing as process' essential component (Factor *Process Capability*).

The *Factor-Criteria-Metrics testability model* shown on the Figure 31 is based on the *testability fishbone* described in detail in [Bi96]. We exclude *Test Suite* (test cases and associated information) from the FCM model. The reason is that the features of a test suite are similar to the characteristics of a *test adequacy* that indicates how well the testing process has been performed, and therefore is not related to the internal quality of the OO product.

The testability measurement may be applied when comparing different adequacy criteria with respect to the ability to reveal hidden faults in the particular software.

5.4.3 Approach

For the purposes of measurement in TROMLAB environment, we focus on the evaluation of the object's testability in terms of controllability/observability of the software units and independently of any particular test strategy. The goal of the design object testability measurement is to evaluate the ease with which the information propagates through messages exchanged between objects - from the external input events to the

external output events (within one period) - in the (sub)system designed.

5.4.4 Testability Measures

The testability of the design object depends on the *controllability* of the object's input and *observability* of its output. Controllability and observability measures quantify objectively the testability in terms of information transfer between objects (information flow), in an abstraction of interactions among the objects of a reactive system. Controllability is measured statically in terms of the quantity of information available on the input to a object. When an object is isolated, all the possible input events can be generated, corresponding to the maximum information content. On the contrary, when the object belongs to an information flow, the information content generated as its input is less because of the loss of information (due to the interactions between the objects). Observability is a static measure that quantifies the information propagated from the object's output events to the final output events. The information quantity that may be received from the object's output at the external outputs is less due to the loss of information.

Testability evaluates the relative contribution of the object's observability and controllability with respect to the maximum loss of information within the information flow. Intuitively, the lower is the contribution to the loss of information, the higher are the objects' controllability and observability, and therefore the higher the ability of the adequate testing to reveal faults. We quantify the loss of information through excess-entropy of the information flow within the system.

For the information flow abstraction purposes, all the information related to the messages exchanged (i.e., events) between objects should be extracted from the design specification. The information flow (corresponding to one period) is originated by external input events and is followed through the communication links between compatible ports of the interacting objects. We construct an object-predicate table whose columns represent port names and rows represent objects in a system design:

$$Object\_Predicate\_Table(object_i, port_j) = \begin{cases} I, & \text{if object}_i \text{ receives Input events at port}_j, \\ O, & \text{if object}_i \text{ receives Output events at port}_j, \\ , & \text{if port}_j \text{ doesn't belong to object}_i. \end{cases}$$

We can view the information flows as an activation of the smallest set of design

objects associated with the (sub)set of external input events. The activation is independent from the remainder of the system. Thus, each information flow partitions the set of design objects into mutually independent subsets. Using Emden's mathematical model, the information flow is quantified by the excess-entropy C on the *Object_Predicate_Table* abstraction of interactions.

Let C_{input} be the excess-entropy calculated on an extraction of the object-predicate table corresponding to the information flow from the external inputs to the object's inputs. Let C_{output} be the excess-entropy calculated on an extraction of the object-event table corresponding to the information flow from the object's outputs to the external outputs. We define the measures as follows:

$$Controllability(Object) = C_{input};$$
$$Observability(Object) = C_{output};$$

An object may belong to several information flows. The object's testability related to only one information flow has been defined formally in [AO2000] and is as follows:

$$Testability_{info-flow}(Object) = \frac{(Observability(Object)+Controllability(Object))}{C_{info-flow}}$$

where $C_{info-flow}$ is the excess-entropy calculated on the abstraction of the interactions within the objects activated by the information flow, and thus logically normalizes the testability measure.

Formally, the object's testability related to several information flows, is defined as

$$Testability(Object) = \sup\{Testability_{info-flow}(Object)\}_{\forall info-flow}$$

Lower value of testability measure indicates higher level of object's testability.

We illustrate the testability computation on the object *Controller1* of the subsystem shown in the Figure 18. The information flow that activates *Controller1* is initiated by the external input event(s) *"Train (1, 2, 3) is approaching the gate 1"*, and partitions the set of objects into two mutually independent subsets. The first subset consists of the objects *Train1*, *Train2*, *Train3*, *Controller1*, and *Gate1*. The second subset consists of the objects *Train4*, *Train5*, *Controller2*, and *Gate2*. The value of C_{max} is calculated from the object-predicate table. In this case we have the following values for the model variables: $n = 9$, $m = 2$, $n_1 = 5, n_2 = 4$. The observability and controllability measures of Controller1 are quantified on related subtables extracted from

the object-predicate table. The values of the measures are obtained by substituting the variables values into the formulas.
The testability measures are theoretically valid, and have to be validated statistically when enough measurement data is gathered.

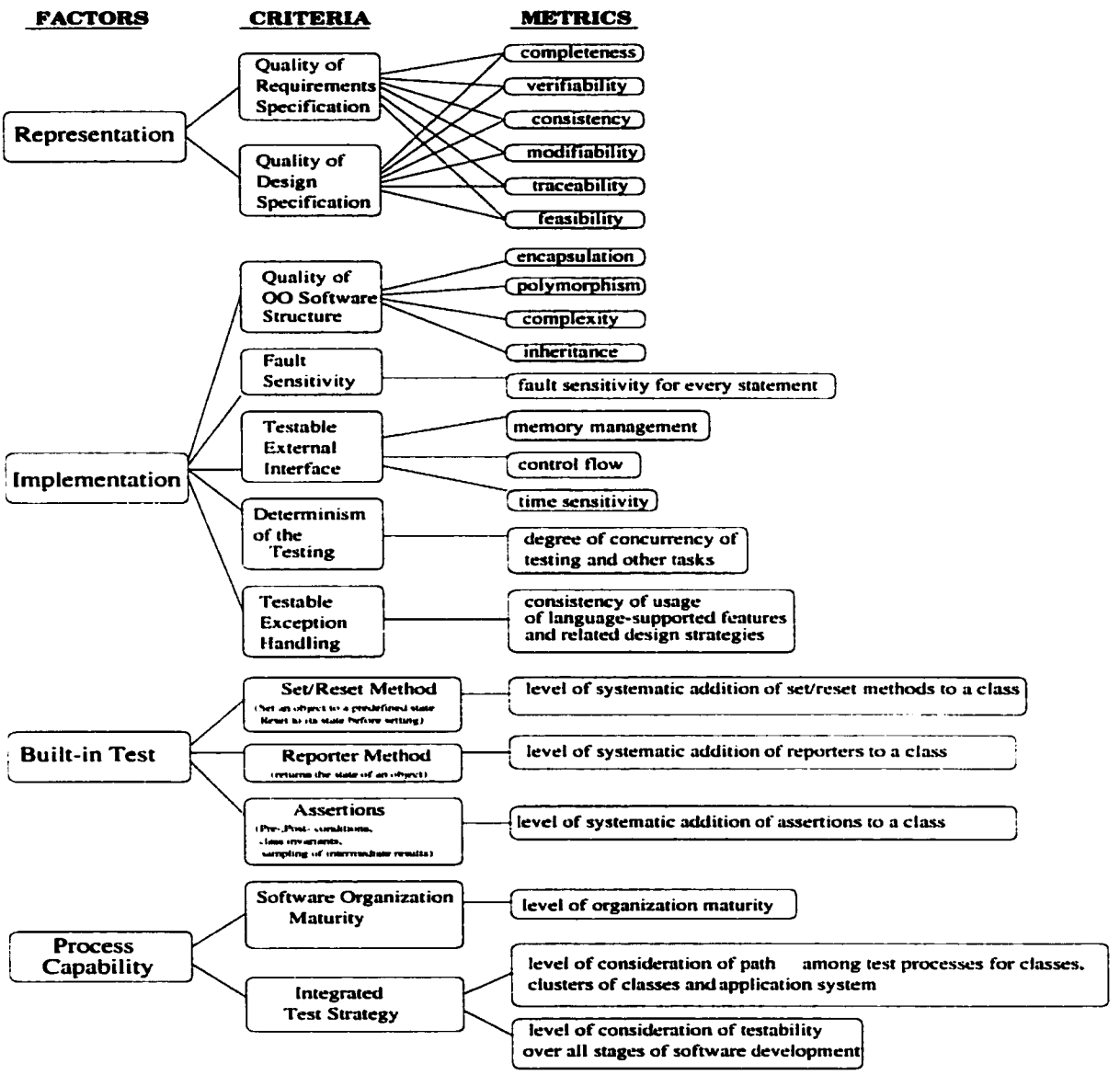


Figure 31: FCM model for Software Testability.

5.5 Functionality Measurement

We define the functional complexity for reactive systems implementation as an amount of work performed in a time slice by the implementation, where the amount of work is understood in terms of quantity of information processed in that period of time, and the number of functions necessary to perform the work.

5.5.1 Approach

Let S be a subsystem of $O_1, \ldots O_n$ reactive objects. Let $Comp(\mathcal{A})$ be set of all computations of a TROM object $\mathcal{A}$ in one period of time (i.e., the time between two consecutive idle states), and $Prot(c_i)$ be the projection of $Comp(\mathcal{A})$ on the signals $(\sigma_1, \ldots \sigma_n)$. Let us define as $Funct(O_i)$ the sequence of signals obtained by merging all protocols ($Prot(c_i)$) on a time coordinate. For the functional complexity measurement purposes, we need the projection of $Funct(O_i)$ on the sequence of events $Events(S) = \{e_1, \ldots e_n\}$. The events present the functions necessary to perform the work in the period of time.

We apply the concepts of information theory to measure the amount of work performed in a time slice by the system in terms of amount of information in the $Events(S)$ sequence. Our version of such a measure is based on an empirical distribution of events within a sequence. The probability p_i of the i^{th} most frequently occurring event is equal to the percentage of total event occurrences it contributes and is calculated as $p_i = \frac{f_i}{NE}$, where f_i is the number of occurrences of the i^{th} event and NE is the total number of events in the sequence. The classical entropy calculation $H = \sum\ p_i\ \log_2\ p_i$ quantifies the average amount of information contributed by each event.

5.5.2 Measure

The functional complexity in a time slice is defined as an average amount of information in the corresponding sequence of events and is computed as follows:

$$FC = -\sum_{i=1}^{n} \frac{f_i}{NE}\ \log_2 \frac{f_i}{NE}$$

The functional complexity in a period of time with higher average information content

should, on the whole, be less complex than another with a lower average information content. The FC measure is intended to be used on the ordinal scale. That is, the FC measure is intended to order the performance of real-time reactive systems in a time period in relation to their functional complexity.

Chapter 6

Test Adequacy Measurement

In this chapter measures for test adequacy are developed. After briefly reviewing the twin concepts of test adequacy criterion and test data adequacy criterion, we develop those measures for real-time reactive systems that are designed and implemented in TROMLAB context.

6.1 Notion of Test Adequacy

To ensure the correctness of the implementation with respect to design, the implementation of the system has to be tested. Informally, the *test adequacy criterion* can be defined as "a predicate that defines software testing objectives in terms of the properties that can be measured". Test adequacy criterion plays two essential roles in any testing method: to specify testing requirements, and to determine the observations that should be done during the testing process. The testing requirements specification has two forms. The first form is called test case selection criterion and is an explicit specification for test case selection. The second form is an explicit specification for test set adequacy measurement when a degree of adequacy in terms of test coverage is associated with each test suite, namely test data adequacy criterion. In particular, the test data adequacy criterion determines whether or not sufficient testing has been done suggesting that the testing can be stopped (*stopping rule*).
To generate a set of test cases from the software product and its own specification, a *testing method* should be defined using a *test case selection criterion*. The level of test case adequacy, which is the degree to which the software is tested, is to be evaluated

as well. The degree of adequacy of testing related to the test adequacy criterion is estimated by the test adequacy measurement. Theoretically valid coverage measures evaluate how well the test suite approximates its target. The measurement of the quality of coverage of the test suite would increase (or decrease) the confidence in tested components.

Zhu [ZHM97] surveys research in test adequacy criteria and provides formal definitions for test case adequacy criterion and test data adequacy criterion. A representational theory of test adequacy measurement and an axiom system to study the properties of test adequacy measures have also been proposed. The evidence of relationships between test adequacy and software correctness, and between test adequacy and software reliability have also been reported. From this work and the work by Fenton (1997), we can conclude that methods on test adequacy criteria can be compared according to fault-detecting ability measurement, reliability measurement of tested software, and test cost measurement.

6.1.1 Test Case Adequacy Measurement

The test case adequacy for untimed systems is measured by the mutation adequacy of a set of test data planting systematically some artificial faults into the program and checking if they are detected by testing. The test case adequacy measure, called *Mutation Adequacy*, is the percentage of programs with artificial faults (mutants) detected, compared to all tested mutants.

The testing method defined according to the test case adequacy criterion generates systematically and efficiently the test cases. The set of these test cases is then considered adequate and no measurement is required. Thus, the test data adequacy measurement is related only to the test data adequacy criterion and, in particular, to the *stopping rule.*

6.1.2 Related Work

The *test data adequacy measurement* associates a degree of test set adequacy according to the *test data adequacy criterion* to indicate how adequately the testing has been performed. There are two important measures associated with every test data adequacy criterion ([FP97]): *test effectiveness ratio* (degree of adequacy of testing) and

minimum number of test cases. The testing effectiveness is defined as $TE = F/E$, where F is the number of faults discovered and E is the effort measured as "person month". To be able to predict the testing time and resources, the testers would need to know the minimum number of test cases needed to satisfy the test data adequacy criterion for a given software. The definition of a measurement formula of calculating the minimum number of test cases depends on each testing approach. A method based on *Prime Decomposition Theorem* is given by Fenton [FP97] to calculate a *minimum number of test cases measurement* for the structural testing approach. The problem of determining *minimum number of test cases measurement* for error-based and fault-based testing approaches still remain an open problem.

The *test effectiveness ratio measure* or equivalently, a *degree of adequacy of testing* of a program p by a test set t with respect to the specification s, according to the test adequacy criterion C, is a function $M_C(p, s, t)$ with value in the interval [01] . To determine whether or not sufficient testing has been done, the test data adequacy criterion as *stopping rule* (or *predicate rule*) should be considered:

> Given a degree R of adequacy corresponding to the minimum number of test cases r, the stopping rule M_R is *True* if and only if the adequacy degree is greater than or equal to R, or *False* otherwise:

$$M_R(p, s, t) = \begin{cases} True, & iff M(p, s, t) \geq r \\ False, & otherwise. \end{cases}$$

The measurement formula of the test effectiveness ratio measure M depends on the specific testing approach and the corresponding test data adequacy criterion. The higher degree of adequacy M indicates more adequate testing of p with respect to s, according to C. Zhu [ZHM97] points out that M depends on the specific testing approach and the corresponding test data adequacy criterion and discusses three approaches to software testing in this context. These are briefly discussed below:

Error-Based Testing This approach focuses on checking critical error-prone software points. An *error* is a defect in the output produced by a software product. Two types of errors are considered:

- *Domain Error* is an error which occurs when the conditions under which the boundaries for a selected sub-domain are incorrect. Here, a domain

refers to an input/output behavior space, partitioned into sub-domains so that the behavior of a software on the points of a sub-domain is equivalent.

- *Computation Error* is an error which occurs due to the incorrect implementation on a given sub-domain.

The *Measure of Adequacy M* is defined as the proportion of errors detected during domain testing and error-based testing, over the total of known errors detected in the software product.

Fault-Based Testing This approach focuses on detecting *faults* (defects in a software). The degree of the fault-based test adequacy M is the ratio of the number of faults found to the total number of faults.

Structural Testing This approach specifies the testing coverage requirements in terms of the structure of the program/specification. The two basic structural testing testing approaches are black-box and white-box testing.

The metrics developed in this section are applicable to all real-time reactive systems, regardless of the notation used for their specification or languages used for their implementation. The metrics will be illustrated for the test cases developed from the formal specifications of reactive systems in TROMLAB environment.

Test case generation methods have recently been reported by Zheng [Zhe02]. Since this testing strategy is specification-based, it is black-box testing of the system. In case of black-box testing approach the test data adequacy criterion is to use a manageable set of test cases to increase the probability of detecting a previously undetected fault while minimizing the probability of detecting the same fault by more than one test case. In this context, the degree of adequacy M indicates a degree of covering of the required functions specified in the formal specifications, by the test data. Reid [R97] reports an experiment comparing the testing effectiveness of the black box testing techniques *equivalence partitioning*, *boundary value analysis* and *random testing* in terms of their degree of adequacy. The experiment considers "all possible input values that satisfy a test technique and all possible input values that would cause a model to fail". The author indicates the need in similar experiment concerning the effectiveness of white-box techniques.

Test data adequacy criteria, as well as adequacy measures related to different test data adequacy criteria can be given for code-based white-box testing of real-time

reactive systems, in a way that are quite analogous to those discussed for untimed systems:

Statement Coverage The adequacy criterion is the requirement to generate test cases to execute every statement in the program at least once. The *Measure of Adequacy M* is defined as percentage of the statements exercised by testing. A reactive program will include constructs for *synchrony*, *delay*, and *concurrency*. The adequacy criterion of untimed systems should be extended to include such constructs in a statement.

Path Coverage The adequacy criterion is the requirement of executing all the execution paths from the program's entry to its exit. A real-time reactive program need not *terminate*. So, this criterion must be changed to

> ... all execution paths from the program's entry to every statement in the program.

The *Measure of Adequacy M* is the percentage of the execution paths exercised by testing.

Linearly Independent Set of Paths Coverage The adequacy Cyclomatic-Number criterion is the specification of restrictions on the redundancy among all paths. The test set contains only v independent paths, where v is a cyclomatic number of the flow graph of the program under test. The *Measure of Adequacy M* is defined as the percentage of the execution of independent paths exercised by testing.

Branch Coverage The adequacy criterion is the requirement of executing all control transfers in the program under test. The *Measure of Adequacy M* is the percentage of the control transfers exercised by testing.

Data-Flow Based Testing In this approach only the data-flow information is taken into account in the definition of the testing requirements. The adequacy criterion is the requirement of executing the flow-graph paths that are significant for the data-flow in the program. The data flow and its timeliness must be part of the criterion. The *Measure of Adequacy M* is defined as percentage of the execution of the required paths.

6.2 Formal Foundation

In this section we discuss a representational theory of test adequacy measurement and an axiom system to study the properties of test adequacy measures. For untimed systems Zhu [ZHM97] has proved the consistency of these axioms. Analytical evaluation of testing techniques considers whether the testing criteria meet adequacy axioms. We justify the validity of these axioms for real-time reactive systems designed and developed in TROMLAB.

Let C_p^s denote a test adequacy criterion C for testing p against s, where $s \in S$ (S is a set of specifications), and $p \in P$ (P is a set of implementations).

Axiom 1. (Inadequacy of Empty Test Set)

$$\forall p \in P \ \ and \ \ s \in S : C_p^s(Empty \ \ Test \ \ Set) = 0$$

This is trivially valid for every system.

Axiom 2. (Adequacy of Exhaustive Testing)

Definition: The program has been *an exhaustively tested* if it is tested on all representable points of the specification domain. Such a test should be adequate independently of the criterion.

$$\forall p \in P \ \ and \ \ s \in S : C_p^s(Exhaustive \ \ Test) = 1$$

This axiom is valid for any system. However, we remark that the behavior of a timed system is in general infinite, thus precluding exhaustive testing. More importantly, it should be noted that test case domain for a timed system with real-time semantics is *dense*, in the sense it is an *uncountable set.*

Axiom 3.(Monotonicity)

$\forall p \in P. \ \ s \in S$, if test set t_1 is a subset of test set t_2, than the adequacy of t_1 is less than or equal to the adequacy of t_2. For real-time reactive systems, the monotonicity prevails in the size of the test set as well as in the granularity of testing. In the test case generation algorithms discussed by Zheng [Zhe02], the state machine description of a reactive class is mapped to a grid automaton with a certain grid size. By increasing the grid size, one may be able to obtain test cases that test the behavior in smaller subintervals of time. Hence, in the later case, test adequacy measure increases. However, it is shown in

Zheng [Zhe02] that the grid size chosen by the grid automaton construction algorithm is sufficient to test the behavior as specified in the time constraints. That is, only when the time constraints change, the grid size will change and thus changing the adequacy measurement. When such a change happens, then it is constrained by the monotonicity property.

Axiom 4. (Convergence)

Let $t_1, \ldots, t_n, \ldots \in T$ be test sets such that

$t_1 \subseteq t_2 \subseteq \ldots \subseteq t_n \ldots$.

Then, $\forall p \in P, s \in S: \quad \lim_{k \to \infty} C_p^s(t_k) = C_p^s(\cup_{k=1}^{\infty} t_k)$

This property is a consequence of monotonicity property. Because the time constraints should be *bounded*, the convergence property holds for test cases generated for testing real-time reactive systems.

Axiom 5. (Law of Diminishing Returns) The more program has been tested, the less a given test set can further contribute to the test adequacy. Formally:

$\forall t_1, t_2 \in T \bullet (t_1 \subseteq t_2 \Longrightarrow C_p^s(t|t_1) \geq C_p^s(t|t_2))$,

where $C_p^s(t|t_1) = C_p^s(t \cup t_1) - C_p^s(t_1)$ Test cases generated by the algorithms discussed by Zheng [Zhe02] test every *observable state and state change*. That is, corresponding to each state in the system specification there a test template is generated, and corresponding to every step of observable state change (as specified by the transition specification) a test template is generated. Test templates can be instantiated with values for parameters and attributes from their respective domains to get test cases for submission to a test execution. These test cases are sufficient [Zhe02] to test the conformance of an implemented program to the specification of the system. Hence, this axiom is valid.

6.3 Approach

Test adequacy measurement for real-time reactive systems have not been widely studied. Results discussed in this thesis are new.

There are two central points when developing a solution for test adequacy measurement: the choice of a formal representation of the domain of the test cases, and a

model of its abstraction. We have chosen to represent formally the test cases domain as a metric space. A metric space is a pair (V, td), where V is a non-empty set and td is a *distance*, or *metric*, such that $td : V \times V \rightarrow R^*$ and the set of distance axioms are satisfied. This approach allows the use of metric-based test case selection algorithm to select the minimal set of test cases (from the set of automatically generated test cases), and metric-based coverage evaluation measurement, both based on the notion of distance between test cases.
$\forall x, y \in V$ the following axioms are satisfied:

Axiom 1. $td(x; y) \geq 0$;

Axiom 2. $td(x; x) = 0 \Leftrightarrow x = y$;

Axiom 3. $td(x; y) = td(y; x)$;

Axiom 4. $td(x; z) \leq td(x; y) + td(y; z)$.

The metric is unique in the sense that there is an order-preserving transformation between two metrics.

6.3.1 Formal Representation and Abstraction of the Test Cases Domain

The test case domain is the set of symbols and terms in the formal specification used to specify the system. The algorithms discussed by Zheng [Zhe02] compute the test cases as follows:

> For a reactive unit A, the set $T_d(A)$ of test cases is computed as
>
> $$X_d(A) \cup Y_d(A),$$
>
> where $X_d(A)$ and $Y_d(A)$ are respectively the state and transition covers of the grid automata $\mathcal{G}_d(A)$ with grid size $d = 1/k$, k being the number of clocks in A. A state cover for a state θ in the grid automaton is a labeled path from the initial state of $\mathcal{G}_d(A)$ to θ. The sequence of labels in a path are the events that take the grid automaton from its initial state to the state θ. An event in the grid automaton $\mathcal{G}_d(A)$ is either an event of A or d.

The label d for a transition from the state θ to a state θ' in $\mathcal{G}_d(A)$ indicates the passage of time at the state s of A, where both θ θ' are mapped to s by the construction of the grid automaton. As an example, the sequence

$$Near?, 1/2, 1/2, In$$

may denote a state cover for the grid automaton of the Train class.

The formal representation that we discuss for developing metrics for test cases are independent of such concrete representations.

In general, let STC denote any test set with any arbitrary representation of test cases in it. Our approach consists in abstracting the elements of STC as binary strings. This would allow the introduction of testing distance as information distance in the space of binary strings. Our choice of information distance is justified by the fact that it is an absolute and objective quantification of a distance between individual objects [Be98].

A two-dimensional array TCA represents the mapping of a test case into a binary string. The definition of the array TCA is as follows:

$$TCA(a, j) = \begin{cases} 1, & if\ test\ case_a\ contains\ event_j \\ 0, & otherwise. \end{cases}$$

Each row of TCA is a mapping of a test case into a binary string. The creation of the array TCA reflects the order of appearance of the events in the test case. The above order then implicitly includes dependence on the ports and time restrictions. Refer to the case study in Chapter 2.

6.4 Testing Distance Measurement

Any distance measurement should satisfy the symmetric and triangle properties for a distance. Intuitively, we expect more similarity between test cases when the distance between the two test cases is small. We want to select test cases for test execution from a test set so that the distance between the selected test case and the set of already exercised test cases is not small.

The distance between two test cases A, $B \in STC$ is abstracted as a distance between their binary string representations $a, b \in V$. The distance between A, $B \in STC$ would depend on two factors, namely, the similarity and the dissimilarity between the test cases. Thus we define the testing distance as:

$$td(a,b) = similarity(a,b) \times dissimilarity(a,b)$$

where $similarity(a,b)$ is defined in terms of the longest common prefix of a and b, and $dissimilarity(a,b)$ is expressed in terms of the minimum amount of change necessary to convert the binary string a into b. The formal quantification models are given below.

Similarity quantification

Let $LCP(a;b)$ be the longest common prefix of the binary string representations $a, b \in V$ of $A, B \in STC$. We define similarity factor between strings as $similarity(a,b) = 2^{-length(LCP(a,b))}$. Note that $LCP(a,b)$ is 0 when there is no common prefix, and $min(length(a); length(b))$ when the longest common prefix coincides with one of the strings. The range of the similarity is between 0 and 1. Higher values indicate lower level of similarity between two test cases and diminish the value of a testing distance. The information distance between two binary strings (elements of a metric space) is computed as the length of the shortest program that translates one string into another.

Dissimilarity quantification

The dissimilarity measure between two binary strings a and b is calculated as the number of elementary transformations that are minimally needed to transform the string $(a \backslash LCP(a,b))$ into the string $(b \backslash LCP(a,b))$. Let us suppose that the abstraction $a \in T$ of some test case $A \in STC$ is the row a of the array TCA. The set of elementary transformations are (1) adding an event j (i.e., setting the value of $TCA(a;j)$ to 1), and (2) removing an event j (i.e., setting the value of $TCA(a;j)$ to 0). The dissimilarity is an unidimensional spatial proximity measure, defined on the ordinal scale. It satisfies the representation and uniqueness conditions for the unidimensional ordinal scale measures and thus is theoretically valid.

Illustration of Testing Distance Measurement

We illustrate the quantification of the distance between two test cases generated [Zhe02] for the Controller-Gate subsystem. Consider the two test cases $A, B \in STC$, $A =$

Test Cases \ Events	Near?	Lower	Down	Exit?	Raise	1/4	1/4	1/4	1/4	Up	Exit
a	1	1	1	1	1	1	1	1	1	1	0
b	1	1	1	0	0	1	1	1	1	0	1

Figure 32: Array Representation of Test Cases

{Near?.Lower.Down.Exit?.Raise. $\frac{1}{4}.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}$.Up}, and B = { Near?.Lower.Down.$\frac{1}{4}.\frac{1}{4}.\frac{1}{4}.\frac{1}{4}$.Exit}. The array representation of the test cases is shown in Figure 32. In this case the values of the model variables are as follows: $LCP(a,b) = 3$; $similarity(a,b) = 2^{-3}$; $dissimilarity(a,b) = 4$; $td(a,b) = \frac{1}{2}$.

6.5 Metric-Based Test Set Selection

Let V denote the set of binary strings representing the original set of test cases STC, ϵ denote the initial target distance, and $\epsilon_{\min}$ denote some comprehensive minimum value of distance such that any approximation on distance smaller than $\epsilon_{\min}$ would not give more meaningful approximations. Let **C** denote some given threshold cost, and *Cost* denote the function representing the resources required to execute the (set of) test case(s). The *Test Selection Algorithm* selects the minimal set of test cases A from the set V. The algorithm stops when the cost limit is reached, the distance $\epsilon_{\min}$ is reached, or there are no more test cases left. We define the distance of a point $t \in V$ from the set A, $A \subseteq V$ by the formula $td(t,A) = inf\{td(t,y) | y \in A\}$.

Test Selection Algorithm

Precondition: $\{V = \mathbf{V} \neq \emptyset \ \wedge \ \epsilon_{\min} > 0 \ \wedge \ C = \mathbf{C} \ \wedge \ A = \emptyset\}$

Step 1. $Initialization(A, V, \epsilon)$

Step 2. $Create - Test - Set(A, V, \epsilon)$

Postcondition: $\{A \neq \emptyset \ \wedge \ (Cost(A) \geq C \ \vee \ \epsilon < \epsilon_{\min} \ \vee \ V = \emptyset)\}$

Algorithm for $Initialization(A, V, \epsilon)$

Precondition: $\{V = \mathbf{V} \ \wedge \ A = \emptyset\}$

Step 1. $t = Longest - test - case(V)$

Step 2. $Add(A, t)$;

Step 3. $Remove(V, t)$;

Step 4. $\epsilon = Length(t) - 1$;

Step 5. **IF** $\epsilon <= 0$ **THEN**

$\epsilon = \epsilon_{min}$;

ENDIF;

Postcondition: $\{A \neq \emptyset \wedge \ \epsilon > 0\}$

Algorithm for

$Create - Test - Set(A, V, \epsilon, \epsilon_{\text{min}})$

Precondition: $\{A \neq \emptyset \wedge \ \epsilon > 0\}$

WHILE

$\neg(Cost(A) \geq C \ \vee \ \epsilon < \epsilon_{\text{min}} \ \vee \ V = \emptyset)$

IF $(\exists$ *test case* $t \ : \ td(t, A) >= \epsilon)$

THEN $Add(A, t)$; $Remove(V, t)$;

ENDIF;

$\epsilon = \epsilon \ - \ 1$;

ENDWHILE;

Postcondition:

$\{A \neq \emptyset \wedge (Cost(A) \geq C \vee \epsilon < \epsilon_{\text{min}} \vee V = \emptyset)$

Algorithm for $Initialization(A, V, \epsilon)$

Precondition: $\{V = \mathbf{V} \ \wedge \ A = \emptyset\}$

Step 1. $t = Longest - test - case(V)$

Step 2. $Add(A, t)$

Step 3. $Remove(V, t)$

Step 4. $\epsilon = Length(t) - 1$

Step 5. **IF** $\epsilon <= 0$ **THEN** $\epsilon = \epsilon_{min}$ **ENDIF**

Postcondition: $\{A \neq \emptyset \wedge \epsilon > 0\}$

Algorithm for $Create - test - set(A, V, \epsilon, \epsilon_{\min})$

Precondition: $\{A \neq \emptyset \wedge \epsilon > 0\}$

WHILE $\neg(Cost(A) \geq C \vee \epsilon < \epsilon_{\min} \vee V = \emptyset)$

IF ($\exists$ *test case* t : $td(t, A) >= \epsilon$) **THEN** $Add(A, t)$; $Remove(V, t)$

ENDIF;

$\epsilon = \epsilon - 1$;

ENDWHILE;

Postcondition: $\{Cost(A) \geq C \vee \epsilon < \epsilon_{\min} \vee V = \emptyset\}$

The test selection algorithm has to be applied in order to select an minimal set of test cases. This minimization would reduce the cost of the testing process while maintaining the same level of efficiency.

6.6 Metric-Based Test Coverage Evaluation

We introduce the metric-based evaluation of test coverage strategy with pre-defined target test sequences which is applicable in safety-critical testing. The basic idea is to choose a test case based upon the test coverage achieved by all preceding test executions. The Law of Diminishing Returns, stated as Axiom 5 in Section 6.2, emphasizes this point. The criterion chosen is the worst-case coverage of some test suite by a selected set A of test cases. The proposed measure of the worst-case coverage of a set V by a selected set A is $CovMax(A) = 1 - m(A)$, where $m(A)$ is the normalized supremum of distances from the set $(V \setminus A)$ to A. The distance between two finite sets $A, B \subset V$ of points in a metric space (V, td) is defined by the *Hausdorff distance* $d_H(A, B) = \max\{sup\{td(a, B) : a \in A\}, sup\{td(b, A) : b \in B\}\}$.

Chapter 7

Reliability Measurement

The *reliability* of a software product is defined in [IEEE90] as the ability of the software to perform its required functionality under stated conditions for specified period of time. The reliability measurement is a procedure that makes predictions about future failure behavior. In this chapter we develop a reliability model for real-time reactive systems whose abstract behavior are modeled by labeled transition systems. We illustrate reliability prediction based upon such a model for Train-Gate-Controller case study.

7.1 Existing Reliability Measures

Several reliability models exist for the object-oriented systems. The common approach to software reliability prediction is the measurement of *remaining failures* based on the collected data on software failures. The most popular reliability models are *Software Reliability Growth Models* (SRGM), that use the failure history of a software obtained during testing and a given operational profile for future prediction. SRG models based on test coverage measurement, are reported in [GO79], [MO84], [Mal94] and [POC93]. The accuracy of the reliability estimate remains as reliability measurement's open problem. Chen at al. [CLW96] points out that the SRGM predictions tend to be optimistic, and propose the pre-processing of the test data (before used by the SRGM) based on both test coverage and testing time.
There exists a verity of definitions and models for software reliability. The reliability measure is defined in [CLW96] as the ratio of the number of failures to the number

of executions in the same operational environment as used in the testing process. The reliability measurement is stated in [ZHM97] in terms of the cost-weighted failure rates.

In [Dyer92] the CLEANROOM reliability certification model is described. The Cleanroom Certification Model is implemented as a PC-based, graphical tool that automates the analysis of statistical usage testing results for the reliability certification process. It takes as input the run times between failures and gives as an output the projected MTTF (mean-time-to-failure) for the next change. The MTTF measure estimates the increase in system reliability as errors are fixed. The testing process stopping criteria is based on comparison of the Markov chain usage model (derived from the specifications) with a Markov chain constructed from the testing experience. Since Markov chains are often very large and complex, Bayesian Belief Networks have been used in [FN2000] to provide reliability predictions. A Bayesian Belief Network is a directed graph whose nodes represent uncertain variables of the reliability model and the arcs are the causal or influential links between the variables. Associate with each node is a set of conditional probability functions that model the uncertain relationship between the node and the parents. Some of the uses of the BBNs reported in [FN2000] have been to provide reliability arguments for critical computer systems, to provide improved reliability predictions of prototype military vehicles, and to predict general software quality attributes such as defect-density and cost.

The measurement of the confidence in the reliability of the software that has passed an adequate test has practical application, especially for safety-critical software when high level of confidence is required. In [Sch96] software reliability predictions approach, used to increase confidence in the reliability of safety critical software (NASA Space Shuttle Primary Avionics Software), is described. The given *Schneidewind Software Reliability Model* predicts the maximum number of *failures* over the life of the software, maximum number of *remaining failures* and the *operational quality* stated in terms of fraction of remaining failures. The prediction of the *test time* necessary to achieve required *operational quality*, as well as the *time to next failure*, are also measured. The above predictions are meaningful for assessing the program manager's decision whether the software is sufficiently tested to allow its release. The model is validated comparing the predictions with actual failure data.

7.2 Approach

This section discusses reliability issues in a rigorous development of real-time reactive systems, and gives measures to predict it in the design phase. Interactions among entities in such systems are complex to describe and reason about. To ensure a level of reliability, it is imperative that the various interactions are understood and their probabilities calculated. Our approach is to model the real-time reactive software design as a Markov system, and propose reliability measurement based on the Markov model.

The reliability assessment method discussed in this work differs from the conventional reliability evaluation methods in important ways:

1. early reliability prediction of real-time reactive system modeled in TROMLAB is new;
2. it is based on the architecture model of the reactive system and the state machine description of reactive units;
3. our approach models the system configuration as a Markov system; and
4. the prediction of the reliability is derived from the steady state of the Markov system.

The approach is illustrated on the railroad crossing problem (Chapter 2). The reliability assessment is discussed in the context of TROMLAB.

7.3 Markov model of the Design

Markov model is one of the most powerful tools available to engineers and scientists for analyzing complex systems. This analysis yields results for both the time dependent evolution of the system and the steady state of the system. The name Markov model is derived from one of the assumptions which allows this system to be analyzed; namely the Markov property. The Markov property states that given the current state of the system, the future evolution of the system is independent of its history. The motivation for applying Markov theory to real-time reactive systems comes from two perspectives: (1) environmental events are not governed by system laws, and

hence may be regarded as random; and (2) the system may execute any one of the several possible transitions in one state to progress to another state.
The operation of the system is represented by a state diagram, which represents the states and rates of a dynamic system. This diagram consists of nodes (representing a possible state of the system, which is determined by the states of the individual components and sub-components) connected by arrows (representing the rate at which the system operation transitions from one state to the other state). Transitions may be determined by a variety of possible events.

7.4 Reliability Model

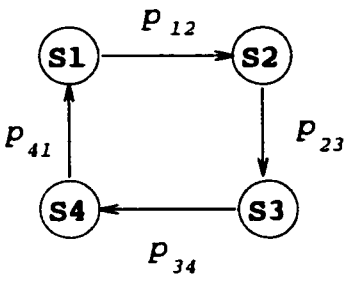
The proposed reliability analysis is based on the software architecture, the state machine models, and the system configuration specification. The advantage is the applicability of the model at design specification phase. The objects, their interactions and the probabilities for the interactions are formalized as a Markov system.

Mapping Reactive Units to Markov Systems We associate with each reactive unit a diagram showing all the states and transition probabilities, and a transition matrix. If an object is in state i, there is a fixed probability, p_{ij}, of it going into state j at the next time step, and p_{ij} is called a **transition probability**. The matrix P whose ij_{th} entry is denoted p_{ij}, is called the **transition matrix** associated with the object. The entries in each row add up to a unity. The transition matrix is calculated as follows:

1. The initial probabilities for all the transitions in the state machine of the reactive object are calculated. The algorithm for calculating such probabilities for a state is based on the following assumptions: 1) all external events that can happen at the state have the same probability; 2) all internal events that can happen at the state have the same probability, and (3) these are in general different.

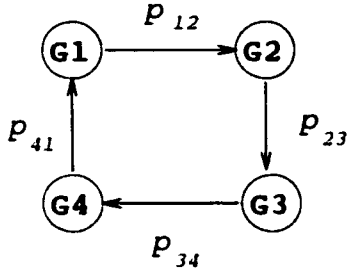
2. In case there is more than one transition $\{l_1 \cup \ldots \cup l_n\}$ of the same type (shared/internal) from state$_i$ to state$_j$, then the above mentioned transitions are substituted by one whose probability is

$$P\{l_1 \cup \ldots \cup l_n\} = 1\ (1 - P\{l_1\}) \times \ldots\ (1 - P\{l_n\})$$



	S1	S2	S3	S4
S1	0	1	0	0
S2	0	0	1	0
S3	0	0	0	1
S4	1	0	0	0

Figure 33: Markov State Transition Diagram and State Transition Matrix for Train Class

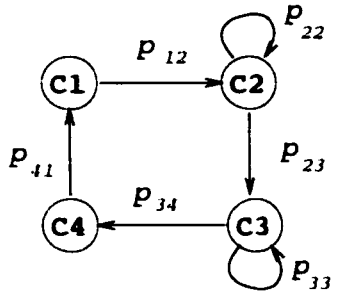


	G1	G2	G3	G4
G1	0	1	0	0
G2	0	0	1	0
G3	0	0	0	1
G4	1	0	0	0

Figure 34: Markov State Transition Diagram and State Transition Matrix for Gate Class

3. The probabilities of all the transitions for a state have to sum to 1.

The approach is illustrated on the Train (Figure 33), Gate (Figure 34) and Controller (Figure 35) classes. The detailed description of the states and transitions for the above reactive classes can be found in Chapter 2, case study.



	C1	C2	C3	C4
C1	0	1	0	0
C2	0	1/2	1/2	0
C3	0	0	3/4	1/4
C4	1	0	0	0

Figure 35: Markov State Transition Diagram and State Transition Matrix for Controller Class

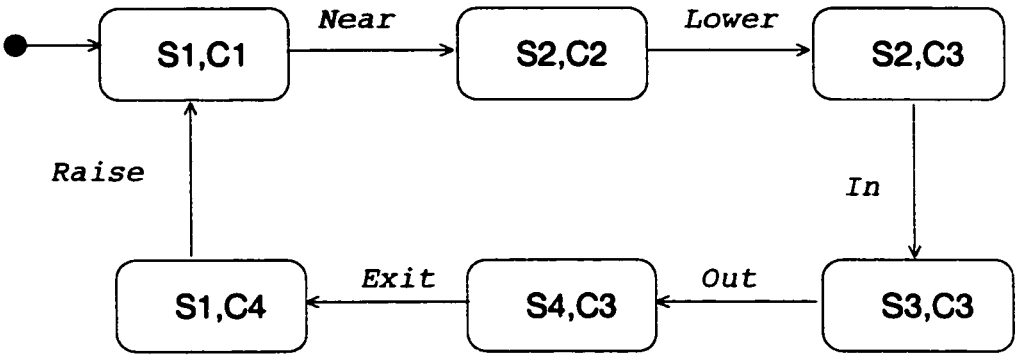


Figure 36: Synchronous Product of Train and Controller (Linear System)

Mapping Object Pairs to Markov System

A system configuration is composed of several synchronously communicating reactive objects. The interaction between two objects is due to shared events. Let P and Q be the state machines for two interacting objects. Each object can be in one or several states, and can pass from one state to another on each time step according to fixed probabilities. The interacting behavior of the two objects is completely described by their synchronous product machine R. Whenever P and Q exchange a message, they change their states simultaneously. The synchronized product of the Train and Controller is shown in Figure 36. The following algorithm creates the synchronous product machine for two given state machines, and calculates the transition probabilities in the product machine.

Algorithm for Transition Matrix for the Synchronous Product Machine

Let E_1 and E_2 be the sets of internal events in the machines P and Q, and F to denote the set of shared events. Let M_1 and M_2 be the transition matrices of P and Q, and R be the synchronized product machine of P and Q. An informal description of the algorithm is as follows:

1. Compute the synchronous product machine.

2. For each state, the transition probabilities of the events in that state are calculated as follows:

   Case 1 All the events are internal, in the sense that every event is internal to one of the machines. Probabilities are obtained by normalizing the

probabilities of events in their respective machines.

Case 2 Only shared events happen in the state. Probabilities are obtained by normalization of the events' probabilities in their respective machines.

Case 3 Both internal and shared events happen in the state. First the probabilities of the shared (external) events are calculated. The justification is that interactions between the objects are due to synchronization events, and therefore they must happen together for the correct execution of the system. Assume that there are only two events e and f, both external, happening in a state. If the external event e has probabilities p_1 in M_1 and p_2 in M_2, and the external event f has probability q_1 in M_1 and q_2 in M_2, then assuming the independence of state transitions in P and Q, the probability of e in the product machine is $p_1 \times p_2$, and the probability of f in the product machine is $q_1 \times q_2$. The probabilities of all the transitions for a state have to sum to 1, therefore the probabilities of the internal events at that state sum to $1 - (p_1 \times p_2 + q_1 \times q_2)$. The above probabilities can be distributed among the internal events at that state according to the weight each internal event has. Note that $1 - (p_1 \times p_2 + q_1 \times q_2) > 0$ because $p_1 \times p_2 + q_1 \times q_2 < (p_1 + q_1) \times (p_2 + q_2) = 1$.

Formal description

Step 1. $p = 1$; // row sum

Step 2. $x_1 = \{e|$ e is a shared event occurring at state i (P) and at state j (Q) $\}$

$x_2 = \{e|$ e is an internal event occurring at state i $(P)\} \cup \{e|$ e is an internal event occurring at state j (Q) $\}$

Step 3. If $x_1 \neq \emptyset$ // calculate probabilities for transitions due to shared events

then $NF = 0$ (Normalization Factor); $set_1 = \emptyset$;

Step 3.1 For each event $e \in x_1$ find the (set of) states i' (P) and j' (Q) such that $i \xrightarrow{e} i'$, $j \xrightarrow{e} j'$

Step 3.2 $y = y \cap \{i', j'\}$, if $\{i', j'\} \notin y$

Step 3.3 $NF = NF + M_1[i, i'] \times M_2[j, j']$

Step 3.4 $M[(i,i'),(j,j')] = M_1[i,i'] \times M_2[j,j']$

Step 3.5 $set_1 = set_1 \cap (j,j')$

Step 4. If $x_2 \neq \emptyset$ // calculate probabilities for transitions due to internal events

then $NF' = 0$ (Normalization Factor); $set_2 = \emptyset$;

Step 4.1 For each event $e \in x_2$, if $e \in M_1$ then

find the state i' (M_1) such that $i \xrightarrow{e} i'$;

$y = y \cap \{i',j\}$ if $\{i',j\} \notin y$;

$M[(i,j)(i',j)] = M_1[i,i']$; $NF' = NF' + M[(i,j)(i',j)]$;

$set_2 = set_2 \cap (i,i')$

else

find the state j' (M_2) such that $j \xrightarrow{e} j'$;

$y = y \cap \{i,j'\}$ if $\{i,j'\} \notin y$;

$M[(i,j)(i,j')] = M_2[j,j']$; $NF' = NF' + M[(i,j)(i,j')]$;

$set_2 = set_2 \cap (j,j')$

Step 5. If $x_1 = \emptyset \wedge x_2 = \emptyset$, the (i,j) row is deleted from M

Step 6. If $x_1 = \emptyset \wedge x_2 \neq \emptyset$

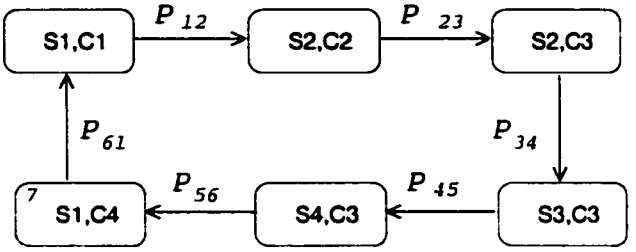
For each $(i',j') \in set_2$ do

$$M[(i,j),(i',j')] = \frac{M[(i,j),(i',j')]}{NF'}$$

Step 7. If $x_1 \neq \emptyset \wedge x_2 = \emptyset$

For each $(i',j') \in set_1$ do

$$M[(i,j),(i',j')] = \frac{M[(i,j),(i',j')]}{NF}$$



	S1,C1	S2,C2	S2,C3	S3,C3	S4,C3	S1,C4
S1,C1	0	1	0	0	0	0
S2,C2	0	0	1	0	0	0
S2,C3	0	0	0	1	0	0
S3,C3	0	0	0	0	1	0
S4,C3	0	0	0	0	0	1
S1,C4	1	0	0	0	0	0

Figure 37: Markov State Transition Diagram and State Transition Matrix for Synchronous Product of Train and Controller in Figure 36

Step 8. If $x_1 \neq \emptyset \wedge x_2 \neq \emptyset$ do

For each $(i^{'}, j^{'}) \in set_2$ do

$$M[(i,j),(i^{'},j^{'})] = \frac{(1 - NF) \times M[(i,j),(i^{'},j^{'})]}{NF^{'}}$$

Step 9. To fill in the matrix M with 0 where there are no entries

Markov model's state transition diagram and state transition matrix for the synchronous product of Train and Controller reactive classes are shown in Figure 37.

Mapping a Subsystem to a Markov System

We differentiate between two cases, that are most commonly occurring system configurations: (1) the system configuration is linear as shown in Figure 38, and (2) the system configuration is non-linear as shown in Figure 41.

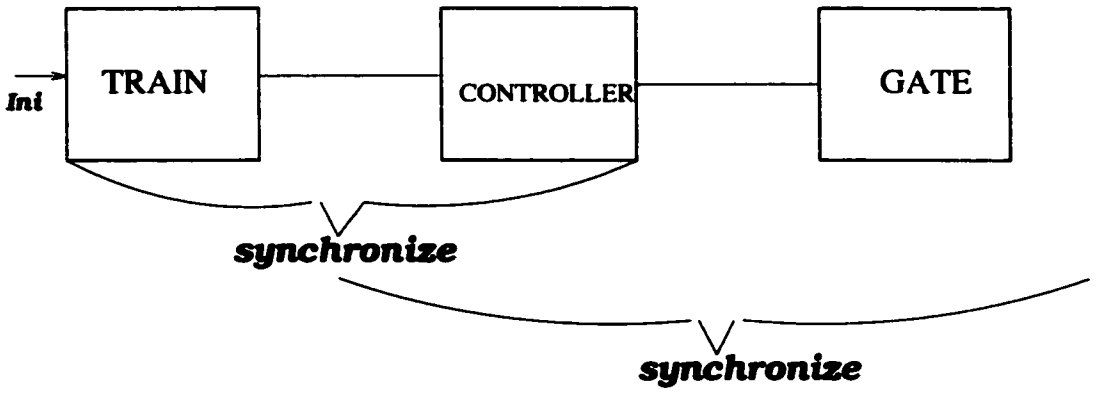


Figure 38: Linear Architecture

Case 1: Linear System

Definition: Linear system is a system whose objects synchronize in the past.
In this case, we extend the algorithm pairwise, i.e.,

1. Apply to a first pair (from the Ini, or start)
2. Substitute the pair with their synchronized machine;
3. While there are more objects do
   - apply the algorithm between the synchronized machine and the next object's machine;
   - substitute the pair with their synchronized machine;

Let M_{Tr} be the Markov model for a Train object; M_C be the Markov model for a Controller object; M_G be the Markov model for a Gate object. Let $\odot$ denote synchronized product operation, and let M_l be the Markov model's matrix for the train-gate-controller linear system $M_{Tr} \odot M_C \odot M_G$. The matrix M_l^k will denote the transition matrix of the linear system for the k_{th} time step.
The case of linear system for $k = 0$ is illustrated on the synchronous product of Train, Gate and Controller (Figure 39). The corresponding Markov state transition diagram and transition matrix are shown in Figure 40.

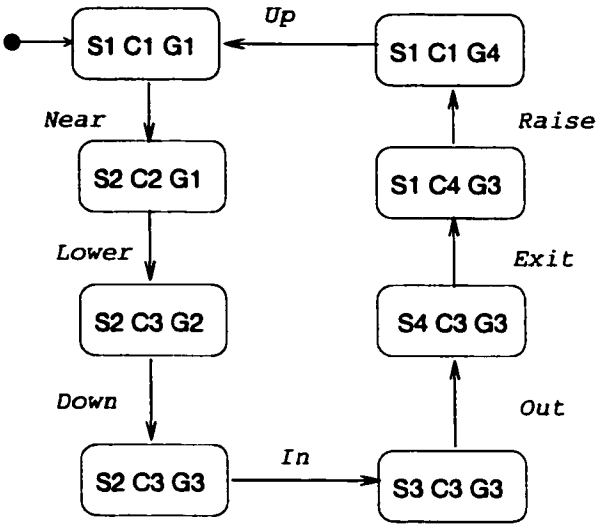
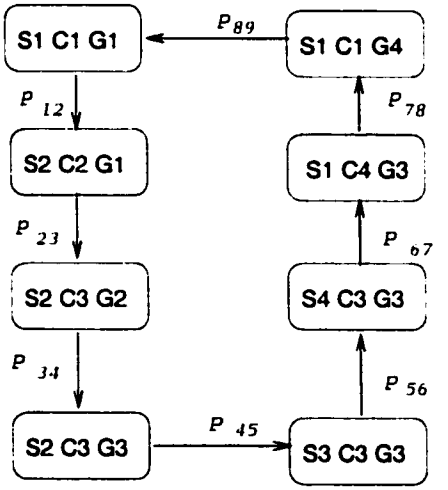


Figure 39: Synchronous Product of Train, Gate and Controller - Linear System



	S1 C1 G1	S2 C2 G1	S2 C3 G2	S2 C3 G3	S3 C3 G3	S4 C3 G3	S1 C4 G3	S1 C1 G4
S1 C1 G1	0	1	0	0	0	0	0	0
S2 C2 G1	0	0	1	0	0	0	0	0
S2 C3 G2	0	0	0	1	0	0	0	0
S2 C3 G3	0	0	0	0	1	0	0	0
S3 C3 G3	0	0	0	0	0	1	0	0
S4 C3 G3	0	0	0	0	0	0	1	0
S1 C4 G3	0	0	0	0	0	0	0	1
S1 C1 G4	1	0	0	0	0	0	0	0

Figure 40: Markov Model for Train, Gate and Controller Linear System Figure 39

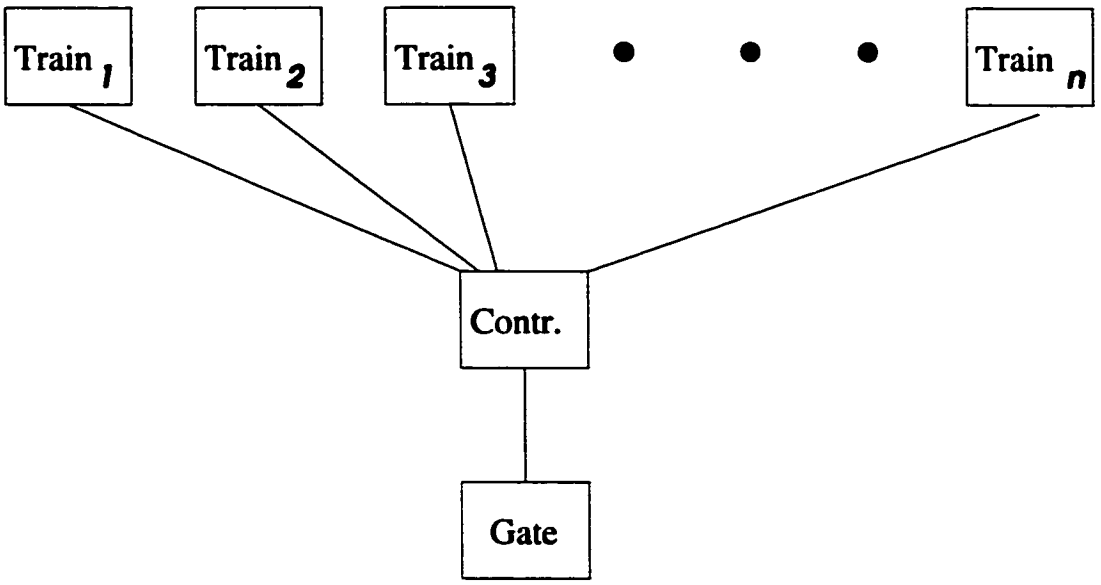


Figure 41: Non - Linear Architecture

Case 2: Non-linear System

We first motivate non-linear system with the example in Figure 41. This diagram shows a system configured with n trains interacting with a controller in order to cross the gate controlled by it.

In case of non-linear system, the synchronous product of Train and Controller shown in Figure 42 would differ from the case of linear system (Figure 36). The synchronous product of Train, Gate and Controller and the corresponding Markov model in the case of non-linear system, are illustrated in Figures 43 and 44.

7.5 Reliability Measures

The reliability prediction for a system configuration composed from n reactive objects is defined as the level of certainty quantified by the source $excess - entropy$:

$$Reliability(Subsystem) = \sum_{i=1}^{n} H_i - H$$

where $H = -\sum_i v_i \sum_j p_{ij} \log p_{ij}$ is a level of uncertainty of the Markov system corresponding to a subsystem; v_i is a *steady state distribution vector* for the corresponding Markov system and the p_{ij} values are the transition probabilities. H_i is a

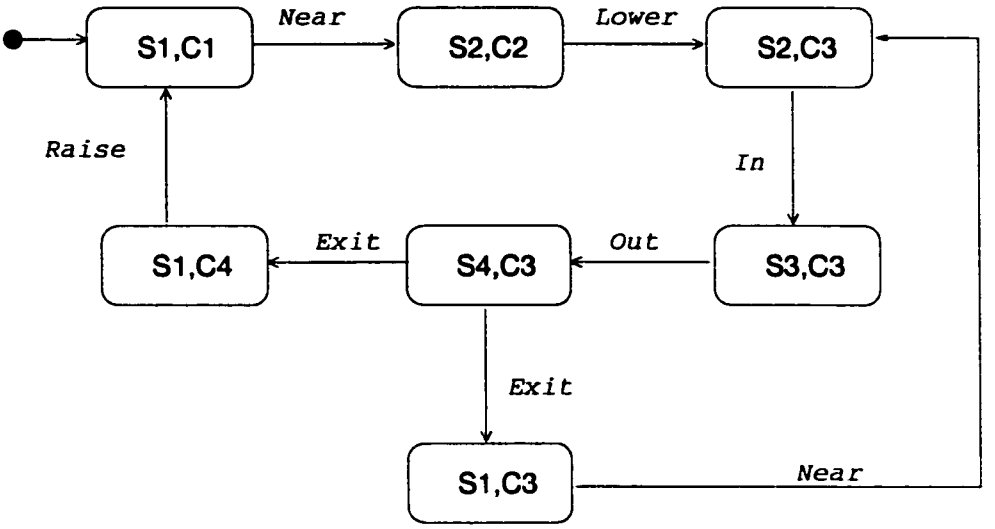


Figure 42: Synchronous Product of Train and Controller (Non-Linear System)

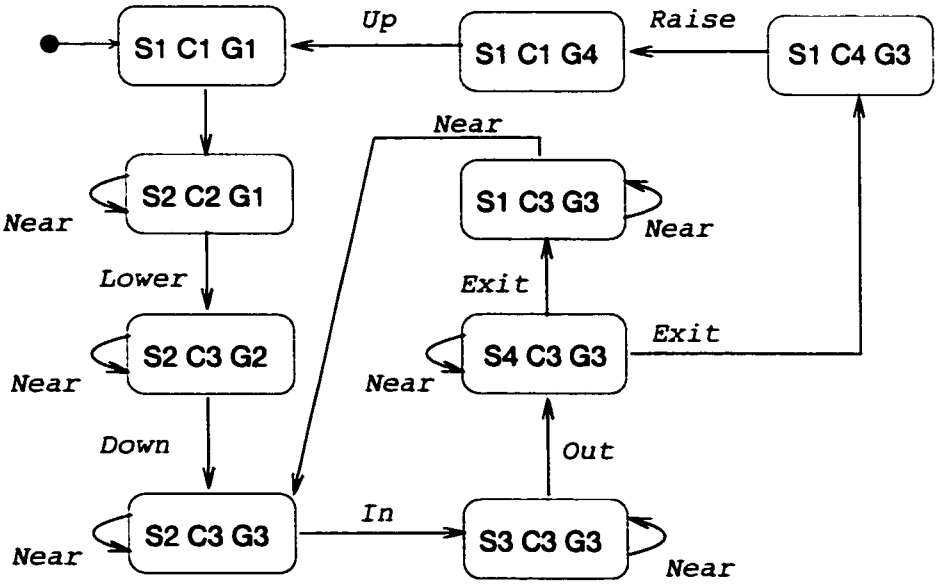
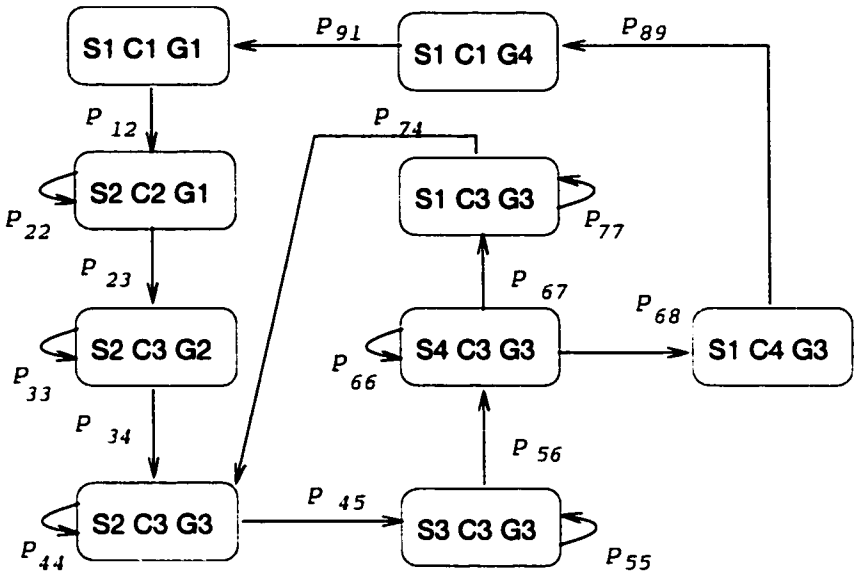


Figure 43: Synchronous Product of Train, Gate and Controller (Non-Linear System)



	S1 C1 G1	S2 C2 G1	S2 C3 G2	S2 C3 G3	S3 C3 G3	S4 C3 G3	S1 C3 G3	S1 C4 G3	S1 C1 G4
S1 C1 G1	0	1	0	0	0	0	0	0	0
S2 C2 G1	0	1/2	1/2	0	0	0	0	0	0
S2 C3 G2	0	0	1/2	1/2	0	0	0	0	0
S2 C3 G3	0	0	0	1/2	1/2	0	0	0	0
S3 C3 G3	0	0	0	0	1/2	1/2	0	0	0
S4 C3 G3	0	0	0	0	0	1/3	1/2	1/6	0
S1 C3 G3	0	0	0	1/2	0	0	1/2	0	0
S1 C4 G3	0	0	0	0	0	0	0	0	1
S1 C1 G4	1	0	0	0	0	0	0	0	0

Figure 44: Markov State Transition Diagram and State Transition Matrix for Synchronous Product of Train, Gate and Controller in Figure 43

level of uncertainty in a Markov system corresponding to a reactive object.

Steady Vector If P is a transition matrix for a Markov system, and if v is a distribution vector with the property that $vP = v$, then we refer to v as a steady state (distribution) vector. To find a steady state distribution for a Markov System with transition matrix P, we solve the system of equations given by

```
x + y + z + . . . = 1
[x   y   z . . . ]P = [x   y   z . . .]
```

where we use as many unknowns as there are states in the Markov system. A steady state probability vector is then given by

```
v = [x   y   z . . . ]
```

H is related exponentially to the number of paths that are "statistically typical" of the Markov system. Thus, higher entropy value implies that more sequences must be generated in order to accurately describe the asymptotic behavior of the Markov system.

We illustrate the calculation of our reliability measure on two configurations of the case study shown in Figures 39 (linear system) and 43 (non-linear system):

$$Reliability(Figure\ 39) = H_{Train} + H_{Gate} + H_{Controller} - H_{Figure\ 39}$$

where $H_{Train} = H_{Gate} = H_{Figure\ 39} = 0$. For calculating $H_{Controller}$ we will need the the steady vector of the Controller: $v_{controller} = \{.125, .25, .5, .125\}$. Then, $H_{Controller} = .25 + .15 = 0.4$. Therefore, $Reliability(Figure\ 39) = 0.4$.

We calculate the reliability for Figure 43 at time step k=0:

$$Reliability(Figure\ 43) = H_{Train} + H_{Gate} + H_{Controller} - H_{Figure\ 43}$$

where $H_{Train} = H_{Gate} = 0$, $H_{Controller} = 0.4$, and

$H_{Figure\ 43} = (v_2 + v_3 + v_4 + v_5 + v_7) \times \log \frac{1}{2} + v_6 \times (\frac{3}{4} \log \frac{3}{4} + \frac{1}{4} \log \frac{1}{4}) > 0$

Therefore, $Reliability(Figure\ 39) > Reliability(Figure\ 43)$. The above measurement data collected on two different configurations for the case study given above, tests the consistency and tracking of the reliability measures.

The reliability prediction for a system is defined as the lowest reliability measure value between its m subsystems:

$$Reliability(System) = min\{Reliability(Subsystem_i)\}_i^m$$

We chose the minimum value due to the safety-critical character of the real-time reactive systems. Higher value of reliability measure implies less uncertainty present in the model, and thus higher level of software reliability.

The Markov model of a configured system changes when the system undergoes change. The calculation of the Markov matrix for the reconfigured system would allow to compare the reliability prediction:

$$Reliability(C_{j-1}) = min\{Reliability(S_i)\}_i^m,$$

where S_i is a subsystem of C_{j-1}, and

$$Reliability(C_J) = min\{Reliability(S_i')\}_i^m,$$

where S_i' is a subsystem of C_j. If $Reliability(C_j) \geq Reliability(C_{j-1})$, then the uncertainty present in the reconfigured system is less than the uncertainty that existed in the current system. The reliability measurement will allow the reconfigured system to be deployed. However, if If $Reliability(C_j) < Reliability(C_{j-1})$, then there is more uncertainty present in the reconfiguration. This would suggest to determine the subsystem(s) of C_J that are responsible for lowering the overall reliability.

Chapter 8

Conclusions and Future Work

During the last fifteen years a great deal emphasis has been put on the quality control of real-time reactive system development. However, not much work was done in the area of quality measurement and management of real-time reactive systems. The results in this thesis seem to be the first ever reported for metrics and measurements of real-time reactive systems. In particular, the quality model for real-time reactive systems, metrics and measurements for design complexity, testing complexity, and reliability are new.

In pursuing this research work, we have focused on the Object-oriented formalism TROM and the framework TROMLAB for illustrating the applicability of our results and the context of tools that can be built on our theory. This does not preclude our results being applied to real-time reactive systems developed in other contexts and formalisms, as long as their semantic basis fits timed labeled transition systems.

8.1 Summary of Significant Results

The two inherent properties of a reactive system are *stimulus synchronization*, and *response synchronization*. In a real-time reactive systems, strict time constraints govern response times as well as internal computations. A critical study of some industrial systems, such as Nuclear Power Plant Control System, reveal that real-time reactive systems also involve concurrency in addition to time-constrained synchronization. Based on such investigation we have proposed factors, criteria, and measures for real-time reactive systems.

Correct measurement and interpretation of measurement results are possible only when the metrics used for the measurement are theoretically sound. We have developed mathematical models appropriate to the object-oriented architecture in TROM formalism and provided methods for measuring the complexity in architecture, maintenance, testability and functionality.

Testing a safety-critical reactive system before deployment is crucial for ensuring the safety of its operational cycle. Theoretically, the test set is only finite, however exercising all test cases is not cost-effective. In order to provide a criteria for stopping the testing process we have developed a test adequacy criteria based on a distance metric on the space of test sequences. We have given an algorithm based on the test adequacy criteria, to select just sufficient number of test cases which are sufficiently dissimilar for a test exercise. The number of selected test cases depends on the distance metric.

Another significant contribution of thesis is a formal approach to calculate the reliability of a real-time reactive system. This approach finds applications in assessing the reliability of time-dependent Web application. In a practical setting, the number of reactive components and their interactions will be large. There are also other factors such as resource constraints, load factor, and communication complexity. From a reliability point of view, we require a good formal model which takes these factors into account. In the formal model proposed in this thesis the load factor and communication delays can be brought in as synchronization constraints, and resources can be modeled within each class (such as the Set in Train class) and timing constraints may be imposed on database transactions. Calculation of transition probabilities for large evolving configurations involves multiplying fairly large matrices. The density of the transition probability matrix of a system depends on the number of transitions in the product matrix, which due to synchronization constraints, might be sparse. The sparsity of the matrix and the availability of very fast powering and multiplication algorithms for matrices may be used to speed up reliability calculation for changing configurations.

One of our goals is to empirically evaluate the reliability model. This is one aspect of our ongoing study in metrics and measurements for real-time reactive systems.

8.2 Future Work

As the measurement procedure is both time and resource consuming procedure, a tool, called *Rose-QA*, for automatic gathering of quality measurement data and analyzing it according to the quality requirements is being designed. It will be integrated with the rest of the tools in TROMLAB framework, shown in Figure 8.
The goal of the quality assistant *Rose-QA* is to assess the quality of the real-time reactive systems design being developed in the TROMLAB environment, before the implementation phase. The quality assistant would provide early feedback on the development process and artifacts, and has to be regarded as an auxiliary tool for providing transparency throughout the development process.
The context of the tool within TROMLAB is defined below:

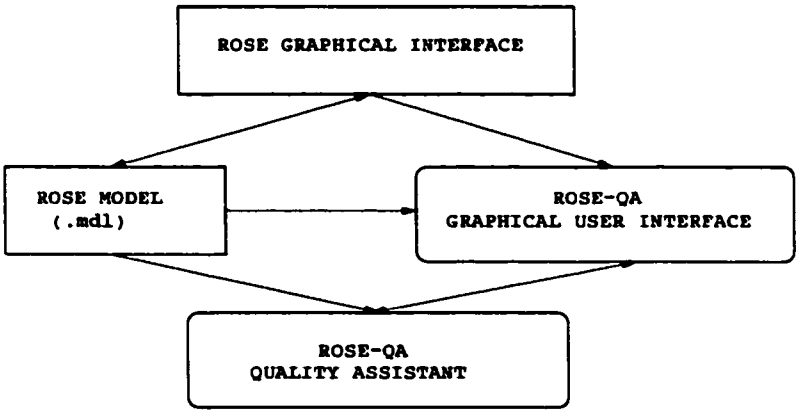


Figure 45: *Rose-QA*: Context

- *Rose-QA* shall run in the Rose environment shown in Figure 45.
- *Rose-QA* shall provide a standard graphical user interface, shown in Figure 46. This GUI will become part of the graphical user interface that already exists in TROMLAB.
- *Rose-QA* shall run only after the execution of the ROSE-GRC Translator [Pop99]. That is, only after the specification of the system is compiled and an internal representation is constructed.

The work reported in this thesis is part of an on-going research project on integrating formal methods with industrial strength methods for the development of real-time reactive system. Important future directions of research include

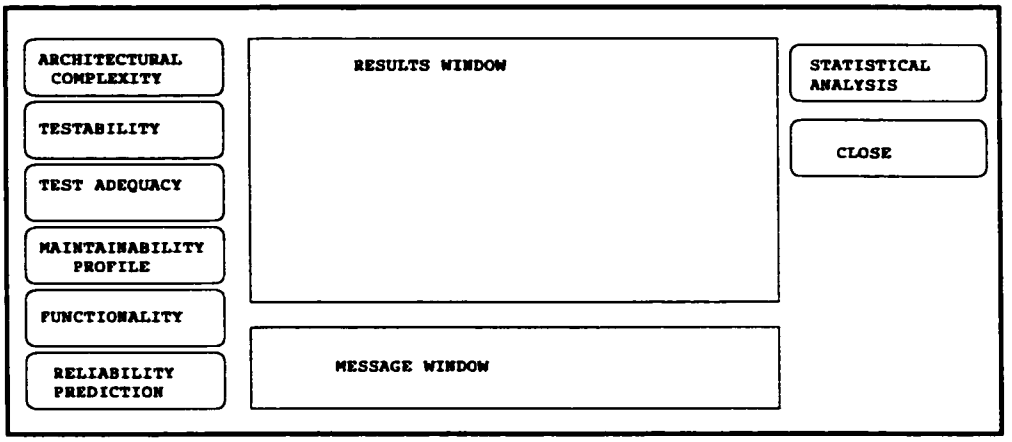


Figure 46: *Rose-QA*: Graphical User Interface

- empirical validation of the measures on large size systems developed in industries,
- development of metrics for formal verification approaches of real-time reactive systems,
- assessing the prediction of the reliability model for evolving safety- critical systems, and
- modeling web-based applications within TROMLAB, assess their quality through *Rose-QA* tool, and empirically evaluate the findings.

Bibliography

[AAM98] V.S. Alagar, R. Achuthan, D. Muthiayen. *TROMLAB: A Software Development Environment for Real-Time Reactive Systems.* (first version 1996, revised 2001), submitted for publication.

[Ach95] R. Achuthan. *A Formal Model for Object-Oriented Development of Real-Time Reactive Systems.* Ph.D. thesis. Concordia University, Montreal, Canada, October 1995.

[All93] J. Allilovic-Curgus. *Metric-Based Theory of Test Selection and Coverage for Communication Protocols.* Canadian Thesis ISBN 0-315-85472-3, 1993.

[AO2000] V.S. Alagar, O. Ormandjieva. *Testing Measurement in Real-Time Reactive Systems.* In Proceedings of ESCOM-SCOPE 2000.

[AOZ2000] V.S. Alagar, O. Ormandjieva, M. Zheng. *Managing Complexity in Real-Time Reactive Systems* In Proceedings of ICESSC 2000.

[AOZ2000] V.S. Alagar, O. Ormandjieva, M. Zheng. *Specification-Based Testing for Real-Time Reactive Systems* In Proceedings of TOOLS-USA 2000.

[AOL2001] V.S. Alagar, Qiaoyun Li, O. Ormandjieva. *Assessment of Maintainability in Object-Oriented Software* In Proceedings of TOOLS-USA 2001.

[AHOZ2001] V.S. Alagar, M. Haydar, O. Ormandjieva, M. Zheng. *A Rigorous Approach for Constructing Reusable, Self-Evolving*

Real-Time Reactive System Software. In Proceedings of CP-WCSE2001.

[BaPS98] F. Brito e Abreu, G. Pereira, P. Sousa. *Coupling-Guided Cluster Analysis Approach to Reengineer the Modularity of Object-Oriented Systems.* Proceedings of the Conference on Software Maintenance and Reengineering 1998.

[BBM96] Basili V., Briand L., Melo W. *A Validation of Object-Oriented Design Metrics as Quality Indicators.* TSE 22(10): 751-761, 1996.

[Be98] C. Bennett, P. Gacs, M. Li, P. Vitanyi, W. Zurek. *Information Distance* IEEE Transactions on Information Theory (44), 4. pp.1407–1423. 1998.

[BG93] I. Bashir and A.L. Goel. *Object-Oriented Metrics and Testing.* In Proceedings of Fifteenth Minnowbrook Workshop on Software Engineering, pages 1-9, Syracuse, New York, Jul. 1993. The Center for Advanced Technology in Computer Applications and Software Engineering(CASE), Syracuse University.

[BK91] R. Banker, R. Kauffman. *Reuse and Productivity in Integrated Computer-Aided Software Engineering: An Empirical Study.* MIS Quartely 15, 375–401, 1991.

[BKWZ94] R. Banker, R. Kauffman, C. Wright, D. Zweig. *Automating Output Size and Reuse Metrics in a Repository-Based Computer-Aided Software Engineering (CASE) Environment.* IEEE Transactions on Software Engineering 20, 3, 169–186, 1994.

[BM99] Benlarbi, W Melo. *Polymorphism Measures for Early Risk Prediction.* In Proceedings of the 21th Int'l Conf. on Software Engineering, 1999.

[BBK78] B. Boehm, J. Brown , J. Kaspar *Characteristics of Software Quality.* TRW Series of Software Technology, Amsterdam, North Holland, 1978.

[Bi96] R. Binder *Testing Object-Oriented Software: a Survey.* Software Testing, Verification and Reliability (6), pp.125–252. 1996.

[Bo81] Boehm B. *Software Engineering Economics.* Prentice-Hall, New York, 1981.

[Ch97] Dennis de Champeaux. *Object-Oriented Development Process and Metrics.* Prentice-Hall, 1997.

[CLW96] M. Chen, M. Lyu, E. Wong. *An Empirical Study of the Correlation between Code Coverage and Reliability Estimation.* Proceedings of the Third International Software Metrics Symposium, pp.133–141, 1996.

[CK91] Chidamber, S., Kemerer, C. *Towards a Metrics Suite for Object Oriented Design.* In Workshop on Processes and Metrics for Object Oriented Software Development, OOPSLA'91, 197–211, 1991.

[CK93] Chidamber, S., Kemerer, C. *MOOSE: Metrics for Object Oriented Software Engineering.* In Workshop on Processes and Metrics for Object Oriented Software Development, OOPSLA '93, 1993.

[CK94] Chidamber, S., Kemerer, C. *A Metrics Suite for Object Oriented Design.* IEEE Transactions on Software Engineering (20), 6, 476–493, 1994.

[CK95] Chidamber, S., Kemerer, C. *Comments on "A Metrics Suite for Object Oriented Design".* IEEE Transactions on Software Engineering (21), 3, 263–265, 1995.

[COTT95] Castell N., Slavkova-Ormandjieva O., Tuells T., Toussaint Y. *Quality Control of Software Specifications Written in Natural Language.* In Proceedings of the 7^{th} International Conference on Industrial & Engineering Applications of A.I. & Expert Systems (IEA-AIE'94), 37–44, 1994.

[CRS96] J Chang, DJ Richardson, S Sankar. *Automated test generation from ADL Specifications.* In Proceedings of the Third International Symposium on Software Testing and Analysis. San Diego, ACM Press, 1996;62-70.

[Dyer92] Dyer, M. *The Cleanroom Approach to Quality Software Development* Wiley, 1992.

[E70] V.Emden. *Hierarchical Decomposition of Complexity.* Cognitive Processes: Methods and Models, pp.361–380. 1970.

[EM97] Ebert C., Morschel I. *Metrics For Quality Analysis and Improvement of Object-Oriented Software.* Information and Software Technology 39, 497–509, 1997.

[FN2000] Fenton N.E. and Neil M. *Bayesian belief nets: a causal model for predicting defect rates and resource requirements.* Software Testing and Quality Engineering 2(1), 48-53, 2000

[FP97] Fenton, N and Pleeger, S. *Software Metrics: A Rigorous & Practical Approach.* Chapman & Hall, 1997.

[FPRSST97] Ferri R., Pratiwadi R., Rivera L., Shakir M., Snyder J., Thomas D. *Software Reuse Metrics for an Industrial Project.* In the Proceedings of Fourth International Software Metrics Symposium, 165–173, 1997.

[F97] Franch Xavier. *The Convenience for a Notation to Express Non-Functional Characteristics of Software Components.* In Proceedings of the First Workshop on the Foundations of Component-Based Systems, Zurich, Switzerland, September 26, 1997.

[FT96] Frakes W., Terry C. *Software Reuse: Metrics and Models.* ACM Computing Surveys 28, 2, 415–435, 1996.

[GH93] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specifications.* Springer Verlag, 1993.

[GO79] A. Goel, K. Okumoto. *Time-Dependent Error-Detection Rate Model for Software Reliability and Other Performance Measures.* IEEE Transactions on Reliability, R-28 (3), pp.206–211, 1979.

[H75] Halstead, M. *Elements of Software Science.* Elsevier N-Holland, 1975.

[Hai99] G. Haidar. *Reasoning System for Real-Time Reactive Systems* Master's thesis, Department of Computer Science, Concordia University, Montreal, Canada, December 1999.

[Hi98] Hislop G. *Analyzing Existing Software for Software Reuse.* The Journal of Systems and Software 41, 33–40, 1998.

[HK81] S. Henry, D. Kafura. *Software Structure Metrics Based on Information Flow.* IEEE Transactions on Software Engineering, SE-7 (5), pp.510–518, 1981.

[HM96] C.Heitmeyer, D.Mandrioli *Formal Methods for Real-Time Computing.* John Wiley & Sons, 1996.

[HP85] Harel D., Pnueli A. *On the development of reactive systems.* In Logic and Models of Concurrent Systems, NATO, Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems. Springer Verlag, 1985.

[HS96] Henderson-Sellers, B. *Object-Oriented Metrics: Measures of Complexity* Prentice-Hall, 1996.

[IEEE83] *An American National Standard IEEE Glossary of Software Engineering Terminology.* ANSI IEEE, 1983.

[IEEE90] *IEEE Standard Glossary of Software Engineering Terminology.* IEEE Std 610.12.1990.

[IEEE93] *Software Quality Factor-Criteria-Metrics Framework.* IEEE Std 1061–1992, 1993.

[ISO91] International Standards Organization. *Software Product Evaluation - Quality Characteristics and Guidelines for their Use.* ISO/IEC Standard ISO–9126, 1991.

[KA94] Khoshgoftaar T., Allen E. *Applications of Information Theory to Software Engineering Measurement.* Software Quality Journal (3), 79–103, 1994.

[Kir94] Kirani S. *Specification and verification of object-oriented programs* Ph.D thesis, Department of Computer Science, University of Minnesota, Minneapolis, MN, November 1994.

[KKKC96] Kim E., Kusumoto S., Kikuno T., Chang O. *Heuristics for Computing Attribute Values of C++ Program Complexity Metrics.* IEEE Transactions on Software Engineering, 104–109, 1996.

[KP96] Kitchenham, B., Pfleeger, S. *Software Quality: The Elusive Target.* IEEE Software (), 12–21.

[LH93] Li W, Henry, S. *Object-Oriented Metrics that Predict Maintainability.* In Journal of Systems and Siftware (23), 111–122, 1993.

[LV2000] L. Lavazza, G. Valetto. *Enhancing Requirements and Change Management.* In Proceedings Fourth International Conference on Requirements Engineering (ICRE 2000), p.106–15.

[MA99] D. Muthiayen, V.S. Alagar. *Mechanized Verification of Real-time Reactive Systems in an Object-Oriented Framework.* Technical Report, Concordia University, Montreal, Canada, 1999, (revised version 2001 submitted for publication).

[Mc76] T. McCabe. *A Complexity Measure* IEEE Transactions on Software Engineering. SE-2 (4), pp.308–320, 1976.

[M77] McCall J., Walters G. *Factors in Software Quality.* RADC TR-77-369 (I, II, III). US Rome Air Development Center Reports NTIS AD/A-049 014, 015, 055.

A Complexity Measure IEEE Transactions on Software Engineering. SE-2 (4), pp.308–320, 1976.

[Mal94] Y. Malaiya, N. Bieman, R. Karcich, R. Skibbe. *The Relationship Between Test Coverage and Reliability.* Proceedings of the Fifth International Symposium on Software Reliability Engineering, 1994.

[MO84] J. Musa, K. Okumoto. *A Logarithmic Poisson Execution Time Model for Software Reliability Measurement.* Proceedings of Seventh International Conference on Software Engineering, pp.230–238, 1984.

[Mut96] D. Muthiayen *Animation and Formal Verification of Real-Time Reactive Systems in an Object-Oriented Environment.* Master's thesis, Department of Computer Science, Concordia University. Montreal, Canada, October 1996.

[Mut00] D. Muthiayen *Real-Time Reactive System Development - A Formal Approach based on UML and PVS.* Ph.D thesis, Department of Computer Science, Concordia University, Montreal, Canada, January 2000.

[Nag99] R. Nagarajan. *Vista - a visual interface for software reuse in TROMLAB environment.* Master's thesis, Department of Computer Science, Concordia University, Montreal, Canada, April 1999.

[NC96] Nesi P., Campanai M. *Metric Framework for Object-Oriented Real-Time Systems Specification Languages.* Journal of Systems and Software (34), 43–65, 1996.

[OKOM97] Obara E., Kawasaki T., Ookawa Y., Maeda N. *Metrics and Analyses in the Test Phase of Large-Scale Software.* Journal of Systems and Software (38), 37–46, 1997.

[PA01] K. Periyasamy and V.S. Alagar. *RTOZ: An Object-oriented Language for the Specification of Real-Time Systems.* Submitted for Publication, March 2001.

[Pai75] Paige, M.R. *Program graphs, an algebra, and their implications for programming. IEEE Trans. Softw. Eng.* SE-1, 3(Sept. 1975).

[POC93] M. Piwowarski, M. Ohba, J. Caruso. *Coverage Measurement Experience During Function Test.* Proceedings of the Fifteenth International Conference on Software Engineering, pp.287–300, 1993.

[Pom99] F. Pompeo. *A Formal Verification Assistant for TROMLAB Environment* Master's thesis, Department of Computer Science, Concordia University, Montreal, Canada, November 1999.

[Pop99] O. Popistas. *Rose-GRC Translator: Mapping UML Visual Models onto Formal Specifications.* Master's thesis. Department of Computer Science, Concordia University, Montreal, Canada, April 1999.

[R97] Reid S. *An Empirical Analysis of Equivalence Partitioning, Boundary Value Analysis and Random Testing.* Proceedings of the Fourth International Software Metrics Symposium, pp.64–73, 1997.

[RB89] Robillard P., Boloix G. *The Interconnectivity Metrics: A New Metric Showing How a Program is Organized.* The Journal of System and Software (10), 29–39, 1989

[RS98] Rine D., Sonnemann R. *Investments in Reusable Software. A Study of Software Reuse Investment Success Factors.* The Journal of Systems and Software 41, 17–32, 1998.

[Sch96] N. Schneidewind. *Reliability and Risk Analysis for Software That Must be Safe.* Proceedings of the Third International Software Metrics Symposium , 1996, pp.142–153.

[Sel99] B. Selic. *Turning Clockwise: Using UML in the Real-Time Domain.* Commun. ACM 42,10 October 1999, pp. 38-54.

[Sh69] Shannon,Claude. *Theory of Communication.* 1969.

[Sh90] Shepperd, M. *Design Metrics: an Empirical Analysis.* Software Engineering Journal 5(1), 3–10, 1990.

[Sch99] N.F. Schneiderwind. *Measuring and Evaluating Maintenance Process Using Reliability, Risk, and Test Metrics.* IEEE Transactions on Software Engineering, Vol.25, No.6, p.769–81.

[Sri99] V.Srinivasan. *Graphical User Interface for TROMLAB Environment* Master's thesis, Department of Computer Science, Concordia University, Montreal, Canada, December 1999.

[Tao96] H. Tao *Static analyzer: A design tool for TROM.* Master's thesis, Department of Computer Science, Concordia University, Montreal, Canada, August 1996.

[TR97] Y. Le Traon, C. Robach. *Predicting Fault Detection Effectiveness.* Proceedings of the Fourth International Software Metrics Symposium, pp.91-98, 1997.

[VMM91] Voas J., Morell L. , Miller K. *Predicting Where Faults Can Hide from Testing.* IEEE Software (March 1991), pp.41-48.

[VM93] Voas J., Miller K. *Semantic Metrics for Software Testability.* Journal of Systems and Software (20), pp.207–216, 1993.

[W86] Weyuker E. *Axiomatizing Software Test Data Adequacy.* IEEE Transactions on Software Engineering (SE–12), 12, pp.1128–1138, 1986.

[W97] S.A. Whitemire. *Object Oriented Design Measurement.* Wiley Computer Publishing, 1997.

[WGS94] E Weyuker, T Goradia, A Singh. *Automatically generating test data from a boolean specification.* IEEE Transactions on Software Engineering 1994; 20(5):353-363

[WKHZKRSP2001] Wijesinha A. *et all. Metric Differences in C++ and Java* In Conference Proceedings of Object Technologies Centers '99, p.387–401.

[Zha00] L. Zhang. *Implementing Real-Time Reactive Systems from Object-Oriented Design Specifications* Master Thesis, Department of Computer Science, Concordia University, Montreal, Canada, 2000.

[Zhe02] M. Zheng. *Automated Generation of Test Suits from Formal Specifications of Real-Time Reactive Systems.* Ph.D. Thesis, Department of Computer Science, Concordia University, Montreal, Canada, 2002.

[ZHM97] H. Zhu, P. Hall, J. May. *Software Unit Test Coverage and Adequacy.* ACM Computing Surveys (29), 4, pp.366–427. 1997.

[ZORS97] Zhuo J., Oman .P, Ramkumar P., Sujay S. *Using Relative Complexity to Allocate Resources in Gray-Box Testing of Object-Oriented Code.* In Proceedings of the Fourth International Software Metrics Symposium, pp.74–81, 1997.