A Framework For Automated Performance Engineering of Distributed Real-Time Systems

by

Hesham M. El-Sayed, B.Sc., M.Sc.

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements of the degree of

Doctor of Philosophy

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario
CANADA, K1S 5B6

October 13, 1999

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

A Framework For Automated Performance Engineering of Distributed Real-Time Systems

submitted by Hesham M. El-Sayed, B.Sc., M.Sc. in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Chair, Department of Systems and Computer Engineering

Thesis Supervisor

External Examiner

Carleton University
October 13, 1999

ABSTRACT

Software Performance Engineering (SPE) is a systematic approach that uses performance models to assess requirements, design and hardware alternatives, starting early in the life-cycle while a wide range of options exists, and continuing through the life-cycle. Despite documented successes of SPE it still faces technical barriers that hinder its widespread use. The first barrier is, that modeling often has a prohibitive cost in time and effort. The second barrier is, that analyzing model results to diagnose problems and recommend design alternatives for performance improvement is not an easy task, as it requires special expertise.

The research proposed here is intended to attack both of these difficulties. It proposes a practical performance engineering framework that provides designers with automated performance assistance for high-level software design. The framework provides assistance to designers by: (1) facilitating the process of building and evaluating performance models, and (2) interpreting model results and recommending design changes to improve designs. The framework integrates several components together to implement a performance engineering toolset which enables practitioners (designers/developers/architects) with little training to establish and maintain performance baselines for complex, real-time distributed systems.

The applicability of the framework to real systems has been tested by an industrial example, and its effectiveness has been evaluated by comparing it with recently published and completely different approaches. The experimental results show that the framework is scalable, in the sense that it can model simple as well as complex systems, robust and has potential to find feasible solution efficiently.

Acknowledgements

I am deeply indebted to my supervisor, Professor Murray Woodside, for his encouragement and friendship throughout this research. His consistent support and guidance have been integral to the success of this work.

I would like to thank Don Cameron for being an excellent mentor and good friend. I wish also to express my gratitude to my friends at the Design Effectiveness group and the members of the Software Engineering and Analysis Lab in Nortel Networks, as well as my colleagues at Carleton University for providing a friendly and encouraging environment.

I am very grateful to my wife, Rania, for her moral support and constant encouragement. She did every sacrifice to cope with my thesis. My daughters, Farah and Rana, have always been my source of great joy and pleasure. My family has been very supportive for me through the years. It is time to let you know how special you all are. Ayman, Sherine, grand mother, uncles, aunts, and cousines. Thank you all.

Financial assistance provided by the Communications and Information Technology of Ontario (CITO), Nortel Networks and Carleton University was greatly appreciated.

Finally, to the soles of my parents, your love and support has made this thesis your accomplishment as much as it is mine. I dedicate my thesis to you.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

1.1 The Problem

Traditional software-design methodologies relegate performance evaluation to the last stages of software design, after the system is operational. For performance-critical systems it is desirable to predict and anticipate performance problems in an early stage of system design, where proper actions can be easily made. Aggressive development schedules, coupled with the inherent complexity of such systems, generally result in inadequate analysis, modeling, specification and tracking of the system's performance properties. Too often the development process proceeds without any real performance baseline. The many design and development decisions are made relatively blind to their performance impact. As the development project matures, performance problems sometimes snowball. The development project then degenerates into an "interactive firedrill" between a waiting customer, systems engineering and the development groups [Strosnider96]. Connie Smith has termed this the "fix-it-later" approach and documented the seriousness of the problems it creates [Smith90]

1.2 The Software Performance Engineering solution

To counter this reactive pattern, Software Performance Engineering (SPE) evolved [Smith90], attempting to understand the performance of software earlier in the life-cycle -- thereby giving designers the opportunity to see the effect on performance of design changes while they are able to do something about it. With SPE, developers build performance into system rather than (trying to) add it later. The SPE techniques use performance models to

provide data for the quantitative assessment of the performance characteristics of software systems as they are developed. Performance modeling of early designs can reduce the risk of performance-related failures by giving an early warning of problems. Performance models provide performance predictions under varying environmental conditions or design alternatives and these predictions can be used to detect problems.

A performance model contains the specification of activities taking place during system responses together with the cost of those activities in terms of execution time as well as the frequency of events requiring response. Solving the performance model yields the total execution time for each concurrent system response together with a utilization number for each CPU in the system. The performance model solution is compared with the requirements for CPU utilization and response deadlines. If the design does not meet requirements, design parameters are changed, the performance model is regenerated and resolved. This cycle continues until requirements are met.

1.3 The barriers that hinder the use of SPE

SPE has evolved over the last 15 years and has been demonstrated to be effective during the development of many large systems [Smith97]. Despite SPE documented successes it still faces technical barriers that hinder its widespread use. The principal problem is the gap between software developers who need the techniques (to have their designs evaluated for performance) and the performance specialists who have the skill to conduct comprehensive performance engineering studies using current modeling tools. Thus, extra time and effort is required to coordinate the design formulation and the design analysis. This limits the ability of designers (developers) to explore the design alternatives.

The primary reason for having this gap has been the lack of a simple, scalable technique for

facilitating modeling of the performance properties of systems design. In the current practice, constructing performance models of complex systems design is labor-intensive, can require significant effort and thus is expensive. To construct performance models, analysts inspect, analyze and translate software descriptions (e.g. design documents or source code) into a model format, which might be a simulation model, queueing model, or a Petri-Net model; consequently, performance models can be expensive to develop and validate, and keeping models up to date with the current state of evolving software systems is also problematic. Accordingly, the models tend to become unwieldy, become insufficiently maintained, so that models are often not used (discarded) and performance is addressed only in the final product. Automated aids are therefore required to ease the process of building performance models.

Apart from the cost of constructing performance models, analyzing performance model results to diagnose performance problems and recommend design alternatives for performance improvement is also not an easy task. Many factors that influence performance are well understood. In practice, however, it can be difficult to interpret the results of performance models to determine which factors are the primary sources of deficient performance, and to find design changes to rectify performance problems. For non-trivial problems, the number of possible design alternatives that must be searched to arrive at an acceptable solution is far too large to be performed manually. It is therefore essential to automate this task. Even if this task were automated, exhaustive search of all possible design changes becomes computationally intractable. Some intelligence must therefore be built into the automation process to guide the search through this potentially enormous solution space for the design improvement problem.

1.4 Implementing Performance Engineering

The research proposed here is intended to overcome some of the difficulties of applying

software performance engineering. It proposes an implementable performance engineering framework that provides designers with automated performance assistance for high-level software design. The framework provides assistance to designers by: (1) facilitating the process of building and executing performance models, and (2) interpreting model results and recommending design changes to improve designs.

The components and interfaces of the framework are depicted in Figure 1.1. The framework has five key components, of which the third and fifth are contributions of this thesis.

1) Scenario Design Paradigm (SDP), a prototype software design technology developed inside Nortel Networks [Jedrysiak94,96], used to speed up the process of constructing SDL functional models.
2) SDT case tool, a commercial design tool [Telelogic96], used to execute SDL functional models and record traces.
3) The Model Builder, a prototype tool produced by this research, used to extract performance models automatically from the execution traces of functional models.
4) The Model Solver, a research tool developed at Carleton University [Franks95], used to solve the performance models and provide performance metrics.
5) The Optimizer, a prototype tool produced by this research, used to interpret performance model results and recommend design changes to improve the design.

The framework integrates all these components together to implement a performance engineering toolset which enables practitioners (designers/developers/architects) with little training to establish and maintain performance baselines for complex, real-time distributed systems.

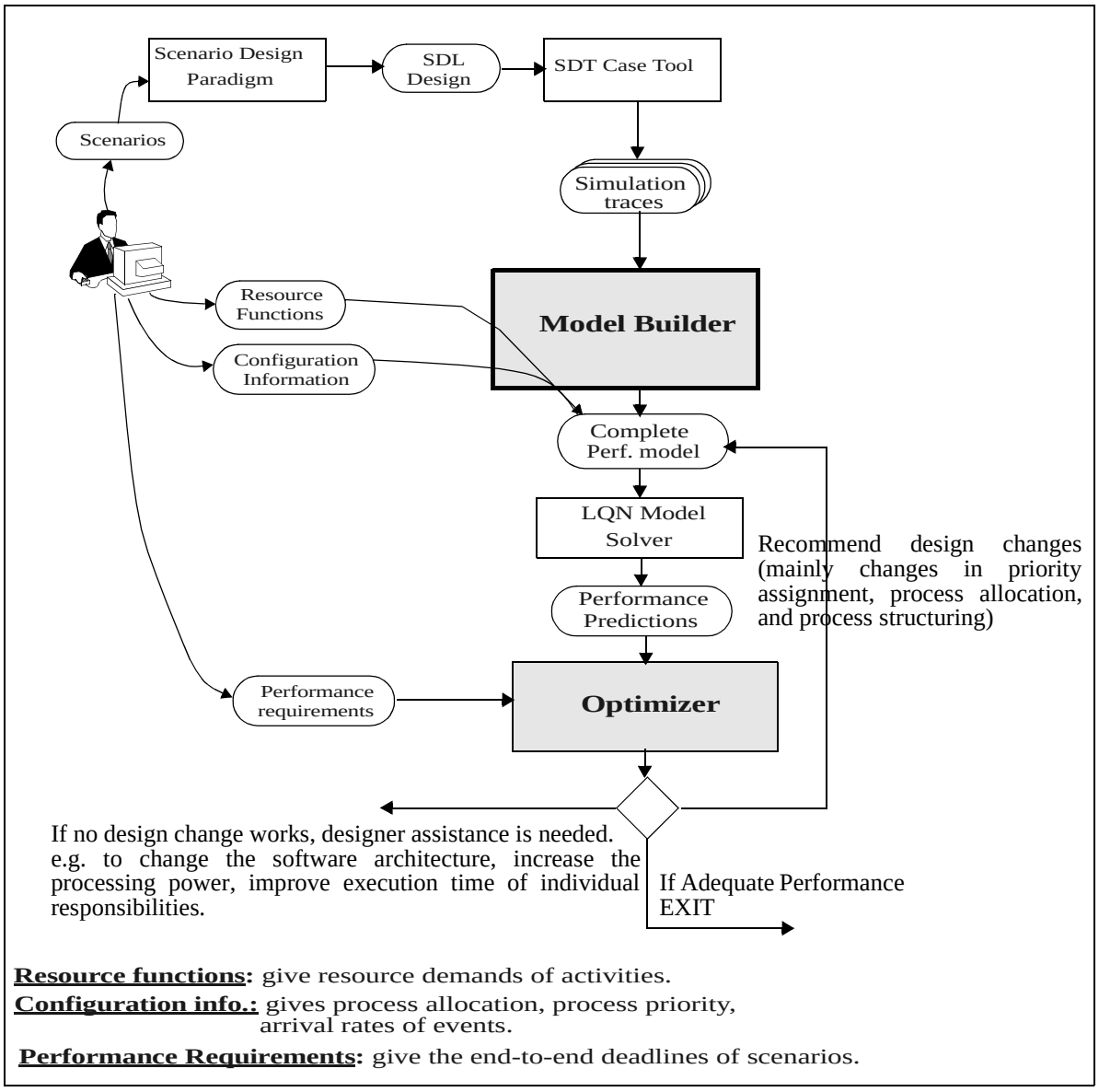


Figure 1.1 Performance Engineering Framework

To connect performance analysis to the software design, the design is first specified in SDL, which may in turn be created from a scenario specification. SDL [ITU93a] is a standardized formal description technique defined for telecom systems and software, based on extended finite state automata. The SDL (functional) model defines the interconnection of operating system processes, the allocation of activities to processes and, when executed, the sequence of activities

for each system response (scenario). The activities of the system are modeled as resource functions that return the cost of doing the activity in terms of execution time. The SDL model is then executed and its traces are analyzed to extract a performance model. The performance model can be solved to get the required performance metrics.

The optimization process includes interpretation of model results, assessment of compliance with user specified performance requirements, identification of suspects of performance problems and recommendation of design changes to alleviate problems. Performance requirements that are considered in this research are the end-to-end response time (deadline) of scenarios. A design is considered to be *compliant* to all requirements (and thus, feasible) if, for every scenario, the probability that response time exceeds deadline is less than some threshold; this threshold can be set to zero in hard real-time systems.

In this research, the design optimization process is formulated as a global optimization problem and an incremental heuristic technique is proposed to solve it. At each step, the design is evaluated. If the design is non-compliant with any of the requirements, possible contributing factors are analyzed, a key factor is determined and some priority changes are recommended. The new configuration is then evaluated and the quality of the solution is estimated. This is repeated until either a feasible solution is found or some stopping criteria are satisfied. If the search on priorities converges without obtaining feasibility, then a structural change is introduced (a design reshaping), and the search is restarted for priorities.

1.5 Origins

From the earlier work in [Hrischuk95], the proposed model builder of this research has retained the concept of angio traces and the goal of model building as well as the outline of the model-building process in the sense that it constructs a performance submodel for each scenario

trace, then merges all these submodels into one model, and finally completes the model by populating it with the information from environment. However, the machinery described here is different from what is reported in [Hrischuk95] and in Hrischuk's later work [Hrischuk99, Woodside98]. In this work the traces are found without building any instrumentation into the CASE tool. So, some important changes have had to be made to the trace processing because of the limited information available in the traces. In addition, the proposed model builder has broaden the target from processes based on RPCs to a system (that is, SDL) based entirely on asynchronous messages, with additional types of execution paths (e.g. parallel execution) and interactions. To deal with the broader problem, considerable extensions to the semantics of the model builder have had to be made. Accordingly, the model-building algorithms have been reconstructed from the ground up, beginning with the trace capture and interpretation. Only the term "angio trace" and the outline of the steps have been retained.

Following the paper of [Hrischuck95], the work of Curtis Hrischuk and the author's were carried out independently. Hrischuk extended the technique in a formal way, applied it to model building based on traces from ObjecTime and described some similar achievements to ours in [Hrischuck99]. The author independently extended the capability of the model-building process to handle asynchronous communications with forks and joins (using different methods), integrated it with a CASE tool (SDP and SDT) and continued into automated design improvements. In any case, the model builder is only the starting point of the thesis, which goes on to tune and/or optimize the design.

1.6 Research contributions

The contributions of this thesis fall into the heads of automation of model-building, optimization, and integration of techniques.

1.6.1 Automation of performance model-building

The first contribution of this research is automation of model-building to make the performance model a simple adjunct to the design (functional) model. The following contributions are particularly notable:

- Inferred angio traces from standard SDL traces (no special instrumentations were built into the design tool).
- Constructed algorithms for trace reduction, including forks and joins (in parallel with C. Hrischuk)
- Demonstrated the use of the model-builder on a number of examples including analysis of Automatic Protection Switching (with 252 processes)

Given the scenarios and the SDL specification, the model can be produced without intervention. This means that it corresponds precisely to the design and can track the development of the design over time. In the SDL tool we use, a final application program can be generated directly from the SDL specification, giving rapid development which is also tracked end-to-end by a performance model.

Automating the performance model-building process gives several advantages: it ensures that the model tracks the design, reduces errors, creates less work for the design team and makes it economically feasible (in terms of developer effort) to do early performance analysis. The cost of constructing functional model is low because of the smaller number of entities represented (processes and responsibilities) and the level of abstraction (amount of information kept for each entity).The construction and solution of the performance model is automatic and thus quicker, cheaper and very low cost.

1.6.2 Optimization strategy

The second contribution of this research is a strategy for optimizing LQN performance models. The following contributions are particularly notable:

- Proposed special diagnostic measures to identify and rank the sources of performance problems.
- Designed an incremental optimization strategy based on identifying critical performance factors and recommending design changes.
- Demonstrated that the optimization strategy is as good as other techniques on standard problems.
- Evaluated the strategy on a large set of randomly generated examples of LQNs.

Automating the interpretation of performance models has several advantages. From the viewpoint of an inexperienced performance engineer or system engineer, the recommendations appear as if provided by a performance specialist. For the experienced performance engineer, the utility of the optimization strategy (the optimizer) lies in its automation of time-consuming data interpretation tasks, freeing the experienced user for higher level tasks.

1.6.3 Integration of techniques

The third contribution is the integration of the above steps with other existing tools to specify scenarios (SDP), to model designs (SDL and the SDT tool), and to solve performance models (LQNS).

1.7 Thesis organization

The rest of the thesis is organized as follows. Chapter 2 briefly presents the background and related work to this research. Chapter 3 describes the approach for specifying designs using

scenarios and SDL and the type and structure of SDL models that are supported by the framework. Chapter 4 describes the proposed methodology for extracting performance models automatically from design specifications. Chapter 5 proposes an optimization strategy for improving a design to meet its performance requirements. Chapter 6 demonstrates the applicability of the complete framework to model and characterize the performance of real industrial systems. It also evaluates the efficiency of the optimization method and compare it with other approaches in the literature which attack similar problems. Finally, the main conclusions of the work presented in this thesis are summarized in Chapter 7 and suggestions for future work are described.

Chapter 2: Background (related work)

The related work to this thesis can be roughly classified into four main areas:

- *Scenario-Driven Software Design*
- *Methodologies which consider performance modeling and analysis early during the design phase (i.e. that integrate design and performance model development)*
- *The Layered Queueing Network Model, which is the model used for performance analysis discussed in this thesis*
- *Aids for design decisions that affect the performance of systems. This includes techniques for process partitioning, process allocation, priority assignment.*

2.1 Scenario-Driven Software Design

Scenario-driven software design is an emerging idea that has the potential to dramatically improve software-system design [Sherman95]. Scenarios are sequences of activities (actions) that specify the desired behavior of a software system. They are slices of the system behavior, which can be considered one at a time in terms of the system model. A scenario diagram (event trace diagram, message sequence chart, interaction diagram) is a graphical formulation of a scenario, specifying how objects communicate with each other and with external actors during the scenario. Each object participating in a scenario is represented by a vertical line; an event is shown as a horizontal arc from the sender object to the receiver(s). Time flows from top to bottom. An example of scenario diagrams (message sequence charts) is shown in Figure 3.6.

Early in the software design process, scenarios are useful for specifying what the designer wants the system to do. They help designers to focus attention on a particular behavior or function thread of the system. Scenarios provide an alternative to trying to mentally juggle the complex interactions of the entire system at once. Scenarios also offer a number of specific benefits over a project's lifetime [Sherman95]:

"• Requirements tracking and tracing: Scenarios are based on requirements. They illustrate how the proposed system will meet the requirements. It is therefore possible to track the requirements during the various phases of a development project by using scenarios.

- Communication: Scenarios offer an excellent vehicle for communication. They are useful for design reviews, customer training, and maintenance. Newcomers to a project can learn about the system with minimal drain on experienced personnel.
- Documentation: Scenarios document the desired behavior. When a design is changed or migrated to another environment, the behavioral description provided by the collection of scenarios directly supports these efforts.
- Design iteration: When a scenario is being defined, the designer's thought processes are stimulated into thinking about what is being presented. Alternative paths, missing parts, and different abstractions are always being considered by creative designers. This may lead to modification of the design when better representations are identified. Scenarios can be refined and expanded as the design evolves.
- Failure cases: Expressing the desired behavior of the system includes situations when incorrect data arrives, resources are consumed, or a time out occurs. Scenarios can be used to illustrate the fail-safe actions of the system under unusual circumstances. This kind of documentation is particularly important in safety-critical or highly dependable applications.
- Test-case definition: Each scenario potentially can serve as the basis for one or more test cases. Depending on the level of abstraction, the scenario details a thread of system activity that's desired and should therefore be tested to verify that it occurs.

- Behavior verification: The scenarios defined at the beginning of the project can be used to compare with event traces captured during simulation runs or actual program execution. Comparison will show areas of mismatch that can be analyzed to determine if any corrections need to be made. Such dynamic testing verifies that the requirements are being met by the implementation. This emphasizes the linkage between the code modules and the requirements.
- Performance improvement: Analyzing a collection of scenarios may reveal paths and nodes that are referenced quite often. This can identify potential bottlenecks that could negatively impact the system performance. The designers are able to use this information to make decisions about rerouting messages traffic or replicating functions to improve the system's overall throughput or performance.
- Impact analysis: Maintainers can take advantage of scenarios by analyzing the interactions between the various parts of the system. They can determine what the overall impact will be if part of the system is modified. This will encourage alternative approaches to be considered, especially if the proposed changes will affect more of the system than anticipated."

The research literature offers an increasing number of scenario-related methods, models and notations. The consideration of concrete system descriptions from a usage oriented perspective - prior to abstract conceptual modeling of function, data, and behavior - has been highlighted in software engineering in the form of use cases within object-oriented analysis and design [Jacobson92]. A number of extensions and alternatives have been proposed, which, e.g., focus on adding structure to use cases [Regnell95,96], on the formal treatment of scenarios [Hsia94], on the use of scenarios during documentation, discussion and evolution of requirements [Potts94] etc. In addition, scenarios are also popular in other fields, most notably human-computer interaction (e.g. [Carroll95]) and strategic planning (e.g. [Blanning95, Bui96]). Scenario use is also becoming a pervasive phenomenon in industrial practice, but comprehensive and expressive studies on the practical relevance of the techniques proposed by

research are still rare. A good survey on scenario usage in system development is reported in [Widen98], and a framework for scenario classification is proposed in [Rolland98].

2.1.1 The use of scenarios in existing software design methods

Use of scenarios is gaining momentum among software methodologists and designers for supporting a number of different design methods. Some of these methods are summarized below:

Object Modeling Technique (OMT): OMT [Rumbaugh91] employs three modeling techniques in software development: object modeling for describing the static relations and properties of objects, dynamic modeling for describing the dynamic behavior of objects and functional modeling for describing the data flow between the processes of the system. Of these models, OMT emphasizes the role of the object model.

Dynamic modeling starts with the construction of scenarios. A scenario is presented graphically as a trace diagram describing the order in which certain events are sent from one object to another. Scenarios are given first for ‘normal’ behavior, and then for ‘exceptional’ behavior. Based on the understanding of the mutual interaction of the objects achieved through scenarios, a state machine (a state diagram in OMT terminology) is constructed for active objects appearing in the scenarios. The OMT method gives no exact procedure for constructing the state machines, but only some informal hints.

Object Oriented Software Engineering, a Use Case Driven Approach (OOSE): Jacobson introduces so called use-case driven design [Jacobson92]. This is software design process based on the idea that the architecture of a system is controlled by what the users want to do with the system. The user’s requirements are modeled by use cases, which roughly

correspond to OMT's scenarios: a use case is a specific way of using a system to accomplish an identified task, consisting of possibly several scenarios. After all of the users (actors) that interact with the system are identified, use cases are defined in words. Emphasis is placed on defining each actor's interaction with the system. Use cases describe how the system performs the requested functions. These scenarios are always depicted from the external user's point of view.

Concurrent Design Approach for Real-Time Systems (CODARTS): CODARTS is an object-oriented software design method for concurrent and real-time systems that emphasizes the structuring of a system into concurrent tasks (active objects). The method uses scenarios and provides support for information hiding (passive objects). In CODARTS, scenarios are included as part of the state-dependent behavioral analysis as described in [Gomaa93]. The main steps are to:

- Build the scenarios, using the state-transition diagram and list of external events.
- Execute each scenario by manually walking through the state transitions for each external event.
- Complete the scenarios, by ensuring that the executed scenarios have driven the state-transition diagram through every state.

The Unified Modeling Language (UML): UML is a general-purpose modeling language for specifying, visualizing, constructing and documenting the artifacts of software systems (in particular object-oriented and component-based systems), as well as for business modeling and other non-software systems [UML99]. It includes many concepts and notations useful for the description and documentation of multiple models, and it enjoys a strong support from academic and industrial communities. UML specifications represents the convergence of best practices in the object-technology industry.

UML allows the description of complex software-driven systems and models through the use of nine different diagram techniques. Each diagram provides a view of a model from the aspect of a particular stakeholder, and each diagram must be semantically consistent with all the others. These diagrams are categorized into two sets. The first set, called behavioral diagrams, focuses mainly of functional and dynamic aspects of systems. It is comprised of five types of UML diagrams:

- Use case diagrams: Show actors and use cases together with their relationships. They describe system functionalities from the user's point of view.
- Sequence diagrams: Describe patterns of interaction among objects, arranged in a chronological order. They originate from Message Sequence Charts [ITU93b].
- Collaboration diagrams: Show generic structure and interaction behavior of the system.
- Statechart diagrams: Show the state space of a given context, the events that cause the transitions of one state to another, and the actions that result.
- Activity diagrams: Capture the dynamic behavior of a system in terms of operations. They focus on flows driven by internal processing.

The second set, called structural diagrams, relates more to components and static characteristics of systems. It includes the following four types of UML diagrams:

- Class diagrams: Capture the vocabulary of a system. They show the entities in a system and their general relationships.
- Object diagrams: Snapshots of a running system. They show object instances (with data values) and their relationships at some point in time.
- Component diagrams: Show the dependencies among software components.
- Deployment diagrams: Show the configuration of run-time processing elements and the software components, processes, and objects that live on them.

2.1.2 Software Design Synthesis from Scenarios

Scenario-driven design is one step along the path toward achieving design synthesis. The goal of design synthesis is to automate the activities within the software lifecycle. In other words, it should turn requirements into designs and then designs into code, and do so reliably and at low cost. Several attempts have been made in universities and industry to achieve automated software design. Some examples are given below.

Scenario Tool: the scenario tool is a a prototype tool for defining and animating scenarios for both structured and object oriented methods [Sherman95], developed at Siemens Corporate Research in Princeton, N.J. The tool was designed so that scenarios can be defined and animated using standard notations (e.g. dataflow diagrams). The tool extends the capabilities of dataflow CASE tools that output files in CASE Data Interchange Format (CDIF) format. The interface to the tool acts as a repository of design diagrams generated by existing CASE tools. These diagrams are read into the tool, and then are displayed in a way in which the scenarios can be annotated and animated. The outputs of the tool are the scenarios that describe the behavior of the system. Animating the scenarios improves scenario communication for purposes of review and training. An area of future research for the authors of [Sherman95] is design synthesis. The idea is to use state-transition information together with a set of scenarios to achieve a semi-automated generation of the state-transition table.

Scenario Design Paradigm (SDP): SDP is a prototype software design technology which has been developed inside Nortel Networks [Jedrysiak94,96]. The Scenario Design Paradigm supports an iterative, incremental, top-down style of system development. It allows designers to express the behavior of a real-time system as a set of concurrent scenarios using a scenario language. Each scenario is a system response to an external event. The active components in the scenarios are messaging entities and the form of their interaction is asynchronous messages. The

scenario language is like a Message Sequence Chart language in textual form but with extended structural syntax that facilitates the translation of the scenario language into SDL. The scenario language compiler performs a kind of state machine synthesis at the process level. It constructs a state machine for each active system component by extracting the behavior of that component from each scenario in which it appears. The state machine language is SDL (Specification and Description Language), a formal description language which is the subject of the International Telecommunication Union Standard Z.100. The compiler produces SDL Graphical Representation (GR) files. These files contain graphical representations of the block architecture as well as the process state machines. Once the system is synthesized into a set of SDL state machines, it can be executed in an SDL CASE tool simulator or on the target product platform. Designers can add scenarios incrementally to the system specification, recompile and execute. SDP has been used in this research to specify scenarios and speed up the process of constructing SDL models. A more complete description with examples will be given in Section 3.2

SCenario EDitor (SCED) [Koskimies94]: SCED is a tool that assists in writing scenario diagrams and producing a state machine that describes the behavior of a particular object, or some method of an object. The tool has been developed at the University of Tampere and Tampere University of Technology, Finland. SCED consists of two CASE components, a scenario editor and a state diagram editor, and a state machine synthesizer, called a generator, integrating scenarios and state machines with various mechanisms.

A cornerstone of SCED is the algorithm that synthesizes a state machine on the basis of a set of scenario diagrams. This algorithm is based on a known machine learning result by Biermann et al. [Biermann76]. SCED has also a number of other features: automated construction of scenario diagrams, automated simplification of state machines using OMT notation, automated layout of state machines, ability to visualize existing systems and produce their abstract behavior

as state machines of their components.

2.2 Integrated design and performance model development

It is a standard concept in engineering to use models to analyze designs. Nevertheless, the integration of performance modeling in the software system design process has not been well studied. Instead, systems design and performance evaluation are often considered as two rather independent areas. As a result of this, each of the two worlds has its own specialists and uses its own models, methods and tools. The major drawback of the use of disjunct models is the extra efforts needed to derive and maintain a separate performance model in addition to the design (functional) model. This effort is often avoided by putting off performance issues as long as possible in the design process. As a result, decisions made blind to their performance impact and problems, and developments ends in an “interactive firedrill” between the customer, systems engineering, and the development groups. The costs of this “fix-it-later” approach have been documented by C.U. Smith [Smith90].

A few authors have considered ways to build a model as part of the design development. The idea of integrating simulation models with system design was used by Zurcher and Randell [Zurcher68] to develop a methodology for the design of computer systems, and was explored by Parnas and Darringer [Parnas67]. Sanguinetti [Sanguinetti79] describes a technique for performance prediction by integrating simulation and software system design using PPML [Riddle72], a system modeling language. Bagrodia and Shen [Bagrodia91] extended the applicability of the integrated approach in significant directions. They proposed an approach, called MIDAS, that supports the design of distributed systems via iterative refinement of hybrid models. A hybrid model is a partially implemented design where some components exist as simulation models and others as operational subsystems. It is an executable model and may be

used to determine the stochastic performance characteristics of a partially elaborated design.

The following subsections presents some recent methods and approaches for integrating design and performance model development. These methods are categorized in the following three groups and discussed in Section 2.2.1 through Section 2.2.4

1) Execution graph-based methods
2) Trace-based methods
3) Language-based methods
4) SDL-based methods

Finally, in Section 2.2.5. a framework for connecting software performance engineering to software design and resource scheduling is discussed.

2.2.1 Execution graph (scenario) based methods

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

Mazzeo et al. [Mazzeo97] described an approach to the specification of concurrent systems which enables a Petri net model of a system to be built up in a systematic way starting from a trace-based specification. The traces are derived manually from system requirements (documents) in the form of CSP specification. A set of rules is then applied to transform the trace-based specifications into a complete Petri net model.

C. Scratchley [Scratchley99] described an approach called PERFECT which evaluates the feasibility of proposed software concurrency architectures for a set of scenarios and a set of quality-of-service requirements. The method first specifies and captures the scenarios using Use Case Maps [Buhr96], which represent paths of execution against a background of the software components that execute them, and annotates the quality-of-service requirements on the scenarios. Then the method allocates subpaths in the specification to processes and decide whether each process will be single or multi-threaded. Finally, an evaluation is performed by constructing and simulating a virtual implementation of the system, which conforms to the specified behavior and the specified concurrency architecture.

2.2.2 Trace-based methods

Automated performance model-building from design prototypes has been tried in [Hrischuck95,99; El-Sayed98]. Hrischuck et al. [Hrischuk95,99] proposed a methodology, called Trace-Based Load Characterization (TLC), for the automatic construction of Layered Queueing Network performance models for message passing distributed and concurrent systems. The basis for this technique are angio traces, which are causal traces of system execution. Model construction in TLC has three steps. It begins by recording angio traces. The trace is analyzed to produce Layered Queueing Network (LQN) sub-model that characterizes the involved processes, their individual activities, and interactions with each other. Lastly, a performance model is generated by merging several LQN sub-models and adding configuration information such as process allocation, the workload on the system, etc.

El-Sayed et al. [El-Sayed98] proposed another method for automated performance modeling from traces. The starting point is a design (in an asynchronous style) expressed in SDL processes, and a set of scenarios. To build a performance model, the SDL model is executed for a set of scenarios, traces are recorded for each scenario, and the model structure and data is

extracted from the traces. A layered queueing model is then constructed.

2.2.3 Language-based methods

D. Menasce and H. Gomaa proposed a methodology to integrate software design and performance modeling [Menasce98, 99]. The methodology is based on a software performance engineering language, CLISSPE. Use Cases were developed and mapped to a performance modeling specification using the language. A compiler for CLISSPE generates an analytic performance model for the system. Service demand parameters at servers, storage boxes, and networks are derived by the compiler from the system specification. A detailed model of DBMS query optimizers allow the compiler to estimate the number of I/Os and CPU time for SQL statements.

2.2.4 SDL-based methods

Several approaches for integrating performance evaluation and formal specification techniques have also been reported in the literature. However, in this thesis, we will focus on approaches to performance evaluation in the context of SDL and MSC. A good survey on these methods is reported in [Thiel99].

Bause et al. proposed an extension of SDL, called Timed SDL [Bause91,93]. The principle is to extend SDL transitions with stochastic and time information. Timed SDL is designed for the examination of qualitative and quantitative system aspects within a single model description. A program package transforms a Timed SDL model into an internal representation of an equivalent finite state machine that can be treated by algorithms for quantitative and qualitative analysis. The solution algorithms employed for performance analysis are of Markovian type.

Martins et al. [Martins95] proposed to extend SDL descriptions by constructs describing

delays, processing resources and workload descriptions. A translation to a performance model that is executable by the tool OPNET (Optimized Network Engineering Tool) is done manually. However, the authors stated that the mapping process from augmented SDL to OPNET can be easily automated. The performance evaluation by discrete event simulation is done in the OPNET environment.

Researchers at University of Essen developed a queuing SDL tool, called QUEST [Diefenbruch95,98]. QUEST is based on the adjunction of time consuming machines that model the congestion of processes due to limited resources. By adding workload models after defining a mapping of workload to machines, an assessable performance model is automatically generated. The description and construction of performance models and their evaluation is supported by the language QSDL (Queuing SDL) and the tool QUEST. The language QSDL provides means for the specification of load, machines and their binding. Load is modeled by QSDL processes by issuing time-consuming requests that are referred for execution to adjunct machines given by queuing stations. QSDL processes are bound to the machines via links and pipes. Processes and machines within the same block are connected with a link. The translation of the QSDL description to an executable simulation program is done automatically.

Mitschele-Thiel et al. [Thiel96a, Thiel96b] described a toolbox called DO-IT toolbox to support performance engineering of SDL/MSC-based systems including model derivation, model-based performance evaluation and optimization. The performance evaluation within the DO-IT toolbox is based on MSC rather than on SDL. An annotated extension of MSC is used to define the performance requirements including the workload, and the resource requirements for specific execution of the system. The performance evaluation techniques provided by the DO-IT toll-box are rather simple and based on deterministic service times. The proposed techniques include bottleneck analysis, critical path analysis and deterministic simulation.

2.2.5 Frameworks for performance modeling

To connect together the various kinds of data relating to performance and design of concurrent software, C.M. Woodside [Woodside95b] proposed a framework called Maps, Paths, and Resources (MPR). It includes a “Core model” which is a kind of conceptual skeleton for the data describing the system, and three views (Maps, Paths, and Resources) which provide the View models. The role of the Core model is to capture the information in each view which is needed for performance engineering in the other views, and to support interview relationships. The role of each separate view is to support engineering analysis of some aspect of the system. The map view shows the structural description of the software; it shows the software components and their interfaces, data, and invocation patterns. The path view shows behavior; it describes the execution path corresponding to various scenarios. Finally the resource view describes the hardware and logical resources for which the software components compete.

2.3 Layered Queueing Modeling

The Layered Queueing Network (LQN) model, proposed by Woodside et al. [Woodside89, Woodside95a, Rolia95], is the model used for performance analysis discussed in this thesis.

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

In Layered Queueing Models, processes represent hardware or software objects which may execute concurrently, and are classified into three categories: (a) client processes: only send messages (requests), and are used to model actual users and other input sources, (b) pure server processes: only receive requests and are generally used to model hardware devices such as processors or disks, and (c) active server processes: can receive requests as well as make their own. They are used to model typical operating system processes.

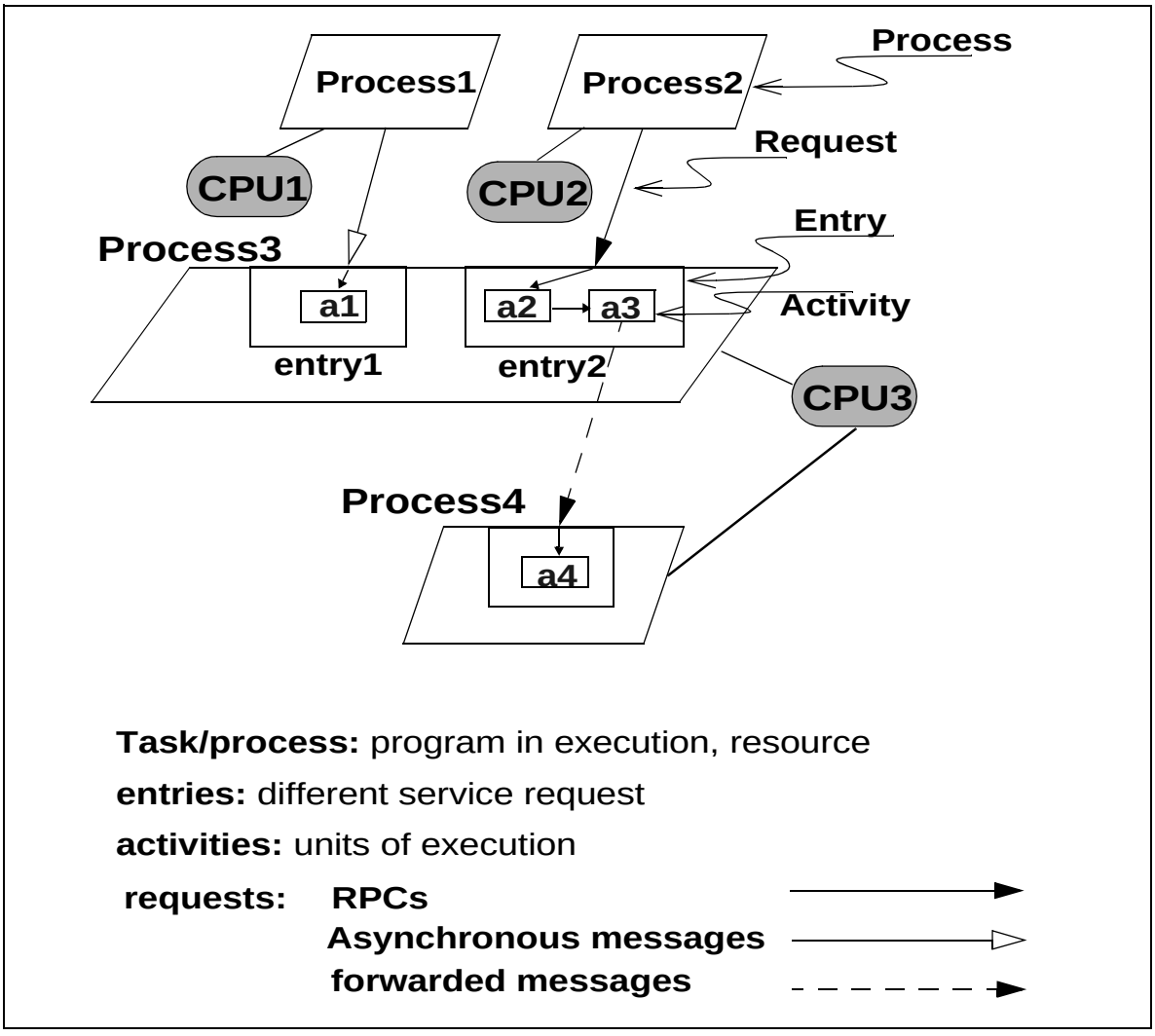


Figure 2.1 An Example of an LQN model

An example of an LQN model is shown in Figure 2.1. In the Figure, software processes are shown as parallelograms and requests from one process to another are made to service “entries”, which are ports or addresses of particular services offered by a process. An entry executes activities with precedence relationships, and activities have resource demands and can make requests to other processes. For each activity, a resource consumption value (for CPU consumption, storage operations, and any other operations of the process to carry out the execution step) must be made available from a repository of “resource functions”. Resource functions are not considered in detail in this thesis, but they are an important part of the information, and in this work they are assumed to be available from previous measurements, or estimated from experience. Requests for service, from one process to another, can be made via three different kinds of interactions:

- RPC or synchronous: sender blocks waiting for a response of its service request and it resumes execution once it receives a reply message from the process that provided the service.
- Asynchronous: sender does not wait for a response from the responding process. No reply message is needed.
- Forwarding: A forwarding request is a special case of an asynchronous request. It occurs when the responding process of an RPC request asynchronously sends the request to another responding process; not to the initiating blocked thread. Each responding process can continue to forward the request further to other responding processes until the last responding process in the series sends a reply directly to the blocked thread.

In a LQN model, there may be several layers of processes interacting with processes from various other layers. The top layer of processes consists of pure client processes referred to as reference processes. Reference processes do not receive requests but only initiate requests. Contrary to reference processes, pure server processes do not initiate requests but only receive

requests. The processes of the middle layers may act as servers and clients.

LQN performance models are used to calculate the throughput of processing, the maximum capacity of service system, and response delays at any level in the system. Analytic tools exist to calculate the mean and variance of delays, and simulation-based tools which can also determine delay distributions and percentiles [Franks95].

2.4 Design decisions that affect the performance

Before we start discussing the design decisions that affect the performance of real-time distributed systems, we will first define the notations used to describe real-time applications (systems).

Real time applications are usually modeled as a set of cooperating transactions (also known as jobs or scenarios). These transactions can be classified, according to their timing requirements, as hard real time (HRT), soft real time (SRT), and not real time (NRT) transactions. A HRT transaction is a transaction whose timely (and logically correct) execution is deemed as critical for the operation of the entire system. Hence, hard real time systems are designed in such a way that deadlines must not be missed. A SRT transaction, instead, is characterized by an execution deadline whose achievement is indeed desirable, although not critical, for the functioning of the system. Hence, soft real time systems are usually not designed to guarantee every single deadline. Instead, the performance of the system is generally measured by the percentage of deadlines it meets which must achieve some specified level. NRT transactions are those transactions which exhibit no real-time requirements.

Application transactions can be further classified as periodic and aperiodic. Periodic transactions are invoked at fixed time intervals. Aperiodic transactions are those whose

invocation can not be anticipated. In either case, the transaction attributes, such as the required resources, the execution time, the invocation period or probability of arrivals, and the end-to-end timing constraints are usually known a priori.

Each transaction can be decomposed into smaller units, called activities. Each activity demands a certain amount of execution time. The execution time of an activity could be its worst-case execution time or its exact execution time if known. The execution order of activities imposes precedence constraints among them. Accordingly, the completion of an activity may enable another activity of the same transaction to be ready for execution. The completion of an activity may also enable multiple activities to run in parallel and this is known as a fork. In addition, an activity may require the completion of several other activities before it can be ready for execution, and this is know as a join.

In the literature, activities are often called processes or tasks and activity graphs are called task graphs. This implicitly implies that, at run time, each activity will be executed by (or mapped to) a different process. In this work, the terms process and task will be used for operating system processes which execute group of activities.

If two processes are assigned to the same processor, communication between them can be achieved via shared memory. Overheads of such communications are usually much smaller than those when the processes reside on different processors, and use interprocessor communication (IPC). IPC introduces processor overhead as well as a communication delay which is a function of inter communication volume and the worst case link delay between the two communicating processors.

The rest of this section will discuss three design decision that can affect the performance of distributed real-time systems, namely:

- ***Process partitioning***, which is concerned with packaging software activities into processes or tasks. Process partitioning has a great impact on performance. It affects the degree of parallelism that can be achieved and also has an effect on priority assignment and process allocation. More importantly, a wrong design decision on process structuring, namely packaging critical and non-critical activities into one shared process, can introduce (bring into existence) a phenomenon that is similar to the priority inversion problem
- ***Process allocation,*** which is concerned with the assignment of processes and data files to processors. Alternative assignment schemes may affect many different performance indicators, including response times, device utilizations, and interprocess communication overheads.
- ***Priority assignment***, which is concerned with assigning priorities to processes. Alternative assignment schemes may affect transaction response times.

2.4.1 Process partitioning

The issue of process partitioning, also called software partitioning, deals with defining concurrent processes from activity graphs. Process partitioning is routinely performed in the development of sophisticated systems. It is necessary, since software processes need to be defined before coding and some other system design activities, such as process allocation, can begin. A process is defined here as the smallest processing entity that is dispatchable by an operating system.

The central problem in process partitioning is that there are at least three objectives which the software architect may require an architectural unit to achieve [Mok86]. The process often serves as an architectural unit for: (1) resource allocation, e.g., the scheduling unit for the CPU, (2) atomicity/recovery management, e.g., a guardian of shared data, and (3) process definition, e.g., to track an airplane in an air traffic control system. These three different aspects may require

the computational requirements of an application to be broken up in different granularity and thereby impose a difficult trade-off in process decomposition. Any one process decomposition may be suboptimal with respect to some of the design objectives, hence the conflicts.

For systems with hard deadlines, A. Mok [Mok84] proposed three strategies for decomposing the computational requirements into a set of processes, and determining their constraints. In the first strategy, which he calls decomposition by critical timing constraints, one process is created to perform the computation associated with each timing constraint. The period and deadline attributes of each process are set to the corresponding parameters of the timing constraint, and a monitor is created to enforce mutual exclusion on the execution of any module called by two or more processes. Decomposition by this strategy is straightforward and the resulting design should be easy to understand. However, this decomposition strategy may yield an inefficient design owing to the unnecessary duplication of some computations in two or more processes with compatible time constraints. This loss in efficiency may be significant since it incurs not only extra computation time but also communication costs for enforcing mutual exclusion.

In the second strategy, which he calls decomposition by centralizing concurrency control, he suggested to group together periodic timing constraints that are compatible with one another and create a periodic process for each equivalence class of compatible periodic timing constraints, and a sporadic process for each asynchronous timing constraint. By clustering as many timing constraints as possible in a single process, both redundant computation and interprocess communication costs can be substantially reduced. The resulting process design can be highly efficient, but difficult to understand or modify. In the third strategy, which he calls decomposition by distributing concurrency control, a periodic process is created for each functional element (activity) in the communication graph, and a sporadic process is created for

each asynchronous timing constraint. Moreover, if a functional element occurs in both a periodic timing constraint and an asynchronous timing constraint, then a monitor is created to enforce mutual exclusion. By assigning a separate process to each functional element, this decomposition strategy has the advantage of maximizing the computations that can be performed in parallel. However, it imposes a higher cost for inter-process communication and context switching. In conclusion, Mok pointed out that there is no single most efficient decomposition strategy. In general, a feasible solution will depend on the trade-off between computation and communication costs. When interprocess communication costs are relatively low, decomposition by maximizing concurrent processes is more likely to succeed. When interprocess communication overhead is predominant, there may be at most one process in a viable design. In between the two extremes lie a wide range of alternatives.

James Huang [Huang85] has done some initial work on software partitioning for distributed hard real-time applications. The objective for software partitioning, in his work, was to satisfy the response time requirements. He points out that it is extremely difficult and perhaps unfeasible to try to measure the quality of software partitioning solutions by means of their potentially achievable response time performance. For instance the response times of a distributed system can not be meaningfully estimated without the processes being allocated, and before the processes can be allocated, they are to be defined first through software partition, leading to a circular dependency, which is not easy to resolve. Accordingly, Huang suggested to approach the software partitioning problem from “the efficient resource utilization” direction to circumvent the difficulty associated with response time estimation. He modeled the software partitioning problem as one that maximizes the partitioning efficiency (efficiency in resource utilization) while observing the constraints on CPU throughput, memory space available, maximally allowed process execution time, and the order of activity execution. The partitioning efficiency (objective) was defined as the ratio of total process execution cost to the sum of total

process execution cost plus total overhead cost. The drawbacks in Huang's model can be summarized as follows: (1) a proper selection of the maximal allowed process execution time can be a difficult task. Process execution time constraint and the partitioning efficiency represent two conflicting requirements. By constraining the process execution time, one can generate more processes. That condition in turn potentially allows a better balanced system load arrangement to be found through the process allocation process. However, with more processes, the overhead costs in process scheduling/dispatch and process communications will increase. (2) The constraint on the order of activity execution is very restrictive. According to this constraint, each activity pair with the preceding as well as succeeding precedence relation is required to be included in the same process. This can lead to an undesirable software partitioning solution.

H. Xin et al. [Xin88] relaxed some of the constraints in Huang's model and presented two heuristic algorithms with different local optimal strategies to find a good solution to the partitioning problem in reasonable amount of time.

V. Iyer et al. [Iyer89] proposed a methodology for structuring software for sequential, pipelined real-time distributed systems. The real-time application was initially represented as an acyclic graph, where the nodes denote the software activities, and the arcs represent the data transmission from a node to its successor nodes. Acyclic graphs are then transformed into trees by replicating nodes that are shared by alternative (decision) paths. The application graph therefore contains only alternative paths, and any single path from the start to the end node may be considered an instance of the application. The application invocation rate is the arrival rate of messages at the start node. Messages are serviced at each node, and sent over a communication link to other nodes. The following information was assumed to be known a prior and must be supplied by the user.

1) Global parameters: number of nodes, link capacity (bytes/second), required

throughput (invocations/second), arrival coefficient of variation, maximum desired node processing power.

2) Node parameters (for each node): input message size, expected execution time (assuming dedicated proc., i.e. no contention), required execution time, coefficient of variation, input flow, message multiplier, number of successor node, flow into successor nodes.

Given the application graph, nodal performance parameters and requirements, the partitioning problem was defined to find the set of node-merged graphs which satisfy the performance requirements while minimizing the total processing power. The partitioning of the application actually involved the coalescence of the nodes of the application graph into a minimum number of clusters (processes). The coalescence of the nodes was carried out by successively combining sequential nodes starting from the start or first node. The size of each process, in time units, was limited by the application execution rate required and the maximum processing power available.

2.4.2 Process allocation

The allocation problem can be stated formally as the problem of allocating processes of an application among processors to achieve some objective, under defined constraints. Based on different objectives, the literature on the allocation problem can be classified into three categories; performance-oriented allocation, schedulability oriented allocation and reliability-oriented allocation. For example, the performance-oriented objective may be to minimize the sum of execution costs and inter-process communication costs, while reliability-oriented objective could be to maximize the system reliability. In each category, the allocation strategy can be static or dynamic. In static allocation all processes are allocated before the application starts to execute. In dynamic allocation the processes are allocated/reallocated at the run time

based on system conditions. In this section we present significant results in the first two categories, but we will focus our attention on the static allocation strategies.

2.4.2.1 Performance-Oriented Allocation

The performance-oriented allocation problem arises when the objective is to minimize (or optimize) a performance-oriented cost function. The strategy chosen to carry out the allocation depends on the performance-oriented cost function. Any values which are measurable in the system can be used as performance indices. For the static allocation problem, the objective functions mostly used are [Hou92]:

- Minimization of total computation and communication times in the system; i.e minimization of the total cost incurred by executing the application on a multiprocessor system. [Stone77, Ma82, Lo88, Houstis90]. In the case of homogeneous system, the objective function reduces to the minimization of the total interprocessor communication time.
- Load Balancing by minimizing the statistical variance of processor utilizations [Bannister83,Tantawi85]. The less the variance, the better the load balance.
- Minimization of the maximum computation and communication times on a processor. The objective function of which was termed the bottleneck processor time in [Chu87], maximum turnaround time in [Shen85], and the system hazard in [Peng89].
- Maximization of an application's bottleneck throughput [Woodside93].

In a general-purpose distributed system, the problem of process assignment is NP-complete [Bokhari81, Lo88]. Hence, the problem of finding a minimum-cost assignment is computationally intractable for all but small systems. Accordingly, several researchers focused on the special cases of the general assignment problem. Stone [Stone77] suggested an efficient optimal algorithm for the problem of assigning processes to two processor (two-processor

problem) by making use of the well-known network flow algorithm in two-terminal network graphs. He showed how the network model can be extended to systems made up of three or more processors. For the three-processor case, Stone and Bokhari [Stone78b] developed an algorithm that finds an optimal assignment. This algorithm works in most cases, but there are pathological cases for which it fails to find an optimal assignment. Stone [Stone78a] also developed an efficient algorithm for the two-processor problem in which the load on one of the two processors is varied. Bokhari [Bokhari79] analyzed the problem of dynamic assignments for two-processor systems and transformed it into a network flow problem under the assumption that all the system characteristics are known for each phase of a distributed application. If the distributed application structure is constrained in a certain way, one can find the optimal assignment in a system of any number of processors in polynomial time. When the structure is constrained to be a tree, the shortest-tree algorithm developed by Bokhari [Bokhari81] yields an optimal assignment. Towsley [Towsley86] generalized Bokhari's results to the case of series-parallel structures. All of the above work is well documented in [Bokhari87] and implicitly assumes the computing system to be fully connected, i.e., there exists a communication link between any two processors. Sager [Sager89] also surveyed other sub-optimal techniques which exploit the shortest path approach. Lee et al. [Lee92] proposed an optimal algorithm to solve the problem of interacting processes to a linear array network of an arbitrary number of processors. A linear array network is composed of linearly-arranged processors, i.e., when any two nonadjacent processors are to communicate with each other, the intermediate processors between them must participate in the communication. Thus, the communication cost per unit information transferred between any two processors increases linearly with the distance between them. The process assignment problem in a linear array network is first transformed into a two-terminal network flow problem, and then solved by applying the network flow algorithm on the related two-terminal network flow graph. This was a direct extension to Stone's network flow approach for the two-processor problem to the case of a linear array of N processors. Also, Lee and Shin

[Lee97] investigated the assignment problem in homogeneous networks in the presence of attached processes. They showed that the assignment problem in an N-processor homogeneous network may be tractable for certain interconnection topologies and not be tractable for others. Their investigation of the problem has led to the development of a modeling technique that is sensitive to the interconnection topology, and transforms the assignment problem to a minimum-cut maximum-flow problem. They applied the modeling technique successfully to solve the problem of assigning M processes in an n-dimensional array and an N-processor tree.

As the general N-processor problem (N>3) in a fully-connected system is NP-complete [Bokhari81]. Hence, several heuristic methods have been proposed to solve the problem. Ma et al. [Ma82] exploited a branch-and-bound method which took into consideration the allocation constraints. Every generated solution was checked first to see whether the allocation constraints are satisfied. If yes, the cost of the solution was evaluated; otherwise, the solution was discarded and future expansion from the solution was eliminated. Arora and Rana [Arora80] considered a heuristic approach to solve the general process assignment problem. The solution was based on the repetitive reassignment of processes, one process at a time. The algorithm terminates when it is not possible to do any further improvement over the current solution.

Chu and Lan [Chu87] proposed a heuristic algorithm to assign processes in distributed systems taking into account precedence relations, communication costs and execution costs. Their algorithm first performed some clustering, following by an exhaustive search for assignments of individual clusters such that the bottleneck (the most heavily utilized processor) is minimized. Lin and Hsu [Lin91] applied the simulating annealing technique to the process assignment problem to minimize the turnaround time.

Sarje and Sagar [Sarje91] considered the minimization of interprocessor communication cost and load balancing as the objective function for process allocation. A heuristic approach

was proposed to find a near optimal solution in polynomial time. Initially, processes that can only be executed on certain processors are assigned. During the assignment process, if a processor was found to be under-loaded, it was favored for the target of the next unassigned process. The algorithm terminates when all processes are assigned.

Woodside and Monforton [Woodside93] proposed the MULTIFIT-COM algorithm, which is a generalized version of the MULTIFIT bin-backing algorithm for allocating processes on a bus-based multiprocessor, taking into account the expected execution and interprocessor communication requirements of the software on the given hardware architecture. The objective of the algorithm is to find an allocation that offers a high system throughput. Throughput was evaluated by an asymptotic bound for saturated conditions and under an assumption that only processing resources are required. The algorithm generalized MULTIFIT in several ways: it uses four heuristic policy choices for initial process sizes, process ordering, intermediate bin weights and process placement. For each of these choices a number of alternatives have been proposed, as shown in Table 2.1, giving 36 potential variations on the algorithm. An evaluation was made on 680 small randomly generated examples. Using all search options, the mean increase in saturation throughput, compared to exact optima found by exhaustive search, was just 1%. Using a carefully chosen subset of just four variations gave almost as good performance as the full set (within 2%).

In this thesis, the MULTIFIT-COM allocator was used to find an initial process allocation to the optimization method, as will be described in Section 5.2 and later in Chapter 6. Although the algorithm can be applied with 36 different policies, only eight policies have been used in this research. The list of these eight policies is shown in Table 2.2. It represents the seven candidates for the reduced-heuristic policy list, which gives about 1.5% degradation [Woodside93], and an additional policy (policy #4), which was recommended by another set of experiments.

Parameters	Options
1. Initial Task Size S (ITS)	*1 : Upper Bound* (S_u) , • execution time plus all potential communications *2 : Lower Bound* (S_l) , • execution time only *3 : Communication Based* (S_c) • only the potential interprocess communication costs.
2. Task Ordering Technique (TOT)	*1 : Absolute (A)* • decreasing order according to task size *2 : Communication-Directed (C)* • special heuristic algorithm [Woodside93]
3. Intermediate Bin Weights (IBW)	*1 : Upper Bound* (W_u) • special heuristic algorithm [Woodside93] *2 : Lower Bound* (W_l) • special heuristic algorithm [Woodside93] *3 : Processing Based* (W_p) • special heuristic algorithm [Woodside93]
4. Placement Criteria (PC)	*1 : Greedy (G), or Largest Fit* • calculate the intermediate bin weights for all possible placements of the task in question. Choose the placement which, to date, satisfies the goal of minimizing the bottleneck workload. *2 : First Fit (F)* • starting at the first bin, calculate the intermediate bin weights. If they are all below the capacity, choose this processor (bin); otherwise move on to the next processor and repeat.

Table 2.1 Heuristic options for the Parameters of MULTIFIT-COM [Woodside93]

Policy No.	Policy Options PO = (PC, TOT, ITS, IBW)
2	(G, A, S_u, W_l)
4 (added)	(G, A, S_l, W_l)
6	(G, A, S_c, W_l)
8	(G, C, S_u, W_l)
10	(G, C, S_l, W_l)
12	(G, C, S_c, W_l)
13	(F, A, S_u, W_u)
19	(F, C, S_u, W_u)

Table 2.2 Candidates for the reduced-heuristic policy list from [Woodside93]

2.4.2.2 Schedulability-Oriented Allocation

The schedulability-oriented allocation problem arises in real-time systems in which the objective is to find an allocation in which the scheduling of the processes meets the specified timing constraints. The problem has been studied by many researchers using different formulations. These formulations represent the different types of real-time systems and the different requirements of the systems. Except for the most trivial cases, the underlying problems of finding an optimal assignment of processes to processors is known to be NP-hard in the strong sense [Garey79]. Since finding an optimal assignment is usually impractical, most researchers have adopted heuristic approaches, which find satisfactory suboptimal process assignments in a reasonable amount of time.

Process allocation in the context of preemptive priority-based Run-time Scheduling.

The problem of process assignment schemes for homogeneous multiprocessor systems

where each processor executes the Rate-Monotonic[1] (RM) scheduling algorithm has been addressed in a number of studies[Dhall78, Davari85]. Typically, the process assignment schemes apply variants of bin-packing algorithms where the set of processors is regarded as a set of bins. The bin-packing problem is concerned with packing different-sized items into fixed sized bins using the least number of bins [Coffman78]. The decision whether a processor is full is determined by a schedulability condition. All of the above assignment schemes are based on the sufficient schedulability conditions for uniprocessor systems derived in [Liu73, Dhall77], and differ mainly in the choice of the bin-packing heuristic.

Dhall and Liu [Dhall78] were the first to propose heuristic algorithms to the problem of assigning a set of independent periodic processes to a minimal number of processors. They proposed two heuristic algorithms, called the Rate-Monotonic First-Fit (RMFF) and Rate-Monotonic Next-Fit (RMNF) algorithms. The schemes are based on the next-fit and first-fit bin-packing heuristic, respectively. In both schemes, processes are sorted in decreasing order of their periods before the assignment is started. Processes are assigned to a so-called current processor until the schedulability condition is violated, in which case the current processor is marked full and a new processor is selected. RMFF first tries to accommodate a process in a processor marked as full before assigning it to the current processor. Dhall and Liu showed that in the worst-case, the assignment produced by the RMFF algorithm uses no more than 2.33 times the optimal number of processors, while RMNF uses no more than 2.67 times. Davari and Dhall [Davari86a] considered another variation of the heuristic, called First-Fit Decreasing-Utilization Factor (FFDUF) algorithm, which improves the worst-case performance to 2 times the optimal number of processors. The FFDUF method sorts the set of processes in non-increasing order of utilization (load) factor. Davari and Dhall [Davari86b] then devised an on-line algorithm, called Next-Fit-M algorithm which has a worst-case performance ratio of 2.2838. The Next-Fit-M

1. The Rate Monotonic algorithm is described in details in Section 2.4.3

algorithm classified processes into M classes with respect to their utilizations. Processors are also classified into M classes, so that a processor in k-class executes processes in k-class exclusively.

Storch and Liu [Storch93] proposed five heuristics for assigning periodic processes with communication costs. Their heuristics are also based on clustering and bin-packing techniques. The goal is to find an assignment of processes to processors such that the total communication cost is minimized, under the constraints that every process is assigned to exactly one processor and the sum of the utilization factors of all processes on any one processor is less than the maximum utilization factor U.

[Buchard94, Buchard95, Oh95] took a different approach for developing process assignment schemes for homogeneous multiprocessor systems where individual processors execute the RM scheduling algorithm. Rather than increasing the level of sophistication of the bin-packing heuristic, they focused on developing tighter schedulability conditions that allow to assign more processes to each processor. They showed that the maximum achievable load on each processor is significantly higher than that suggested by previous work.

Mutka and Li [Mutka95] used a branch-and bound algorithm for periodic process allocation. They also assumed that each processor would serve the processes assigned to it using the RM algorithm. The algorithm traverse the process assignment tree in a depth-first fashion to allocate the processes to the processors and to determine if the given allocation can meet the periodic deadlines. If the algorithm determines that a process assignment exceeds the bounding condition, then it backtracks up the tree one level and searches depth-first with a new untested branch. If it is determined at one level that the bounding conditions are not exceeded, the search continues down the current path. When the search has successfully traversed to a leaf of the tree, a feasible process assignment is found. The algorithm begins searching for feasible schedules

using the single inequality test [Liu73] as the bounding condition at each node in the process assignment tree. If no feasible schedules can be found, then the algorithm uses a less pessimistic test, the multiple inequality test [Mutka95], as the bounding condition. Likewise, if no feasible schedules are found, then the algorithm uses the least pessimistic test, the numerical test [Lehoczky89], as the bounding condition.

Tindel et al. [Tindel92] used a simulated annealing algorithm to solve the allocation and scheduling problem combined. In this study, a distributed rate-monotonic algorithm was also used as the scheduling scheme.

Process allocation in the context of Pre-run-time (static table driven) Scheduling.

While run-time scheduling has the virtues of dynamic reconfiguration and high processor utilization, the goal of satisfying the critical timing constraints can not be guaranteed. One feasible approach to solve such a problem is to schedule every process a priori using the table-driven approach. In table-driven scheduling paradigm, a table is constructed for each processor, using some heuristic, to identify the start and completion time of each process.

Ramamritham [Ram95] proposed an algorithm for allocating and scheduling periodic real-time transactions across sites in a distributed system. The algorithm takes as input the graphs depicting each periodic transaction in the system, and produces as output process allocations and a schedule on each processor. The algorithm consists of two parts. The first part of the algorithm decides whether a cluster of communicating activities of a transaction should be assigned to the same site. This decision is based on the computation times of the activities in a cluster and the amount of communication between them. More specifically, if the fraction $\left(\frac{\text{sum of the computation time of the activities}}{\text{cost of communication}}\right)$ is lower than a tunable parameter called

communication factor, CF, then the activities should be assigned to the same site. Given the clustering done in the first part, the second part assigns the clusters of activities to the sites in a system and also determines a feasible schedule, if possible. Since the first part of the algorithm eliminates some of the communication (by deciding that certain activities should be assigned to the same site), the search space in the second part is considerably reduced.

Several authors have used branch-and-bound techniques to solve the assignment (allocation and scheduling) problem. Peng and Shin [Peng89] considered the problem of assigning and scheduling a set of communicating periodic transactions to a set of heterogeneous processors so that the maximum transaction response time is minimized (i.e. minimize the system hazard). They proposed two branch-and-bound algorithms, one for process assignment and the other for scheduling. Hou and Shin [Hou92] addressed the problem of assigning and scheduling the activities of periodic transactions, subject to precedence and timing constraints, in a distributed real-time system. Their branch-and-bound technique finds an assignment that maximizes the probability of meeting transaction deadlines.

Coli and Palazzari [Coli95] described a method to map (i.e. allocate and schedule) an application with some real-time constraints into a parallel system. They formulated the mapping problem as a minimization problem, defining a cost function whose minimization leads to the optimal mapping of the application into the parallel system. The searching space over which the minimization must be carried out was defined; this space encloses all the feasible allocation and scheduling modalities for the application in the parallel system. The minimization was carried out through a simulated annealing algorithm. The presented algorithm ignored the interprocessor communication costs and the topology of the parallel system.

2.4.3 Priority Assignment (RT-Scheduling)

According to priority-driven scheduling, every released but not completed process instance has a priority. At any time, the process instance with the highest priorities gets to execute. A process instance will not give up the processor until it completes or is preempted by another process instance with a priority higher than the current executing one.

Among priority-driven scheduling schemes, fixed-priority scheduling has gained popularity due to the simplicity in implementation, the ease of schedulability analysis and its predictable behavior during transient overload situations. According to fixed-priority scheduling, all the instances of the same process have the same priorities, and the priority of each process instance never changes from its release until its completion (i.e. the priority of a process remains fixed once it is assigned). Because of this reason, it is often said that a priority is assigned to a process rather than to process instance. The schedule of the execution of the processes is determined by assigning different priorities to processes. If a set of processes can be scheduled such that all process deadlines can be met by some algorithms, then the process set is said to be feasible. In this section, we focus our attention to fixed-priority scheduling algorithms.

The solution to the problem of optimizing the assignment of priorities is well known for uniprocessors. Liu and Layland [Liu73] proposed the Rate-Monotonic algorithm (RM). The RM algorithm assigns priorities to processes according to their periods, where the priority of a process is in inverse relationship to its period. In other words, a process with a shorter period is assigned a higher priority. They also proved the optimally of rate monotonic priorities for sets of independent periodic processes with the deadline of each process equal to its period. The algorithm is optimal in the sense that no other fixed priority assignment algorithm can schedule a process set which can not be scheduled by the RM algorithm. Leung and Whitehead [Leung82] proposed the deadline monotonic policy for processes having deadlines less than or equal to their

respective periods. With this policy, priorities are assigned in a similar manner to rate-monotonic: the shortest deadline process is assigned the highest priority; processes with successively longer deadlines are assigned successively lower priorities. The deadline-monotonic priority assignment is equivalent to rate-monotonic priority assignment when, for all processes deadline equals to period. Deadline-monotonic priority assignment is optimal in a similar manner to rate-monotonic: if there exists a feasible priority ordering over a set of processes, a deadline-monotonic priority ordering over those processes will also be feasible. Audsley [Audsley91] provided an algorithm that was capable of finding a solution (if one existed) to the problem of assigning priorities when one or more of the deadlines were larger than the process periods.

Rate-monotonic and deadline-monotonic priority assignments assume that all processes share a critical instant (i.e. common release time). If processes are permitted to have arbitrary offsets then this condition may not hold. Under these circumstances neither priority assignment policy is optimal. Indeed, whilst rate-monotonic and deadline-monotonic priority assignments can be achieved in polynomial time, Leung et al questioned whether the same applied to priority assignments for processes with no common release time [Leung82]. Later, Audsley showed that optimal priority assignment can be achieved by examining a polynomial number of priority orderings over the process set, assuming an exact (pseudo-polynomial) feasibility test [Audsley93]. In 1990, Lehoczky et al showed that neither rate-monotonic nor deadline-monotonic priority assignments are optimal if process deadlines exceed their periods [Lehoczky90]. However, the optimal priority assignment algorithm given by Audsley also applies to such processes.

In multiprocessor or distributed systems the problem of priority assignment is much more complicated, because of the strong interaction between the response times of actions in the

different resources. The problem was proved to be an NP-hard problem [Betatti94]. Since no efficient optimal algorithms have been found for any NP-hard problem and exhaustive solutions are impractical for all but extremely small systems, most researchers focused on heuristic solutions to the priority assignment problem. Heuristic solutions to the priority assignment problem can be classified into three categories: rate-based methods, deadline-based methods and optimization-based methods [Sun96].

2.4.3.1 Rate-based methods

The rate-monotonic priority assignment for single processor systems can be straightforwardly applied to end-to-end systems, by assigning to every process a priority inversely proportional to the period of its parent transaction. Obviously, this method is not optimal. As a matter of fact, the performance of this method is in general inferior to other heuristic methods [Sun96].

2.4.3.2 Deadline-based methods

Kao and Garcia-Molina [Kao93] have studied the deadline-based priority assignment in the context of a soft real-time system. They presented multiple strategies for distributing end-to-end deadlines over sequential [Kao93] and sequential/parallel [Kao94] transactions. The impact of priority assignment in real-time databases was investigated in [Lam96, Purimetla94]. Jun Sun [Sun96] adopted some of Kao's strategies and proposed two more strategies for priority assignment in hard real-time systems.

According to (based on) the deadline-based methods, a relative deadline is first assigned to each process, then a priority is assigned to a process inversely proportional to its relative deadline. This means, a higher priority is assigned to a process with lower relative deadline.

Different priority assignment methods compute the relative deadlines in different ways. For example:

- In the Global-Deadline method [Sun96], also called Ultimate-Deadline in [Kao93,94] the relative deadline of the process is the end-to-end (global) deadline of the parent transaction, to which the process belongs.
- In the Effective-Deadline method [Sun96, Kao93,94], the effective deadline of a process is equal to the global deadline of the transaction minus the total execution time of its successors.
- In the Proportional-Deadline (PD) method [Sun96], the proportional deadline of a process is obtained by dividing the global deadline of its parent transaction among its processes proportionally to their execution times.
- In the Normalized-Proportional-Deadline (NPD) method [Sun96], the relative deadline assignments is similar to that of the PD method, except that the processor utilization is taken into account. In the PD assignment, if two processes of the same transaction have equal execution times, they will have equal proportional deadlines. In the NPD assignment, however, if one process executes on a more heavily utilized processor than the other, it is assigned a relatively longer deadline.

2.4.3.3 Optimization-based methods

Tindel et al. [Tindel92] treated the problems of priority assignment and process allocation combined, as a discrete optimization problem. They used a simulated annealing technique to find a feasible priority assignment and a feasible load partition at the same time.

Garcia et al. [Garcia95] proposed a heuristic iterative algorithm, called HOPA, that given an initial local deadline assignment, finds an improved solution in reasonable time. For each iteration a new deadline assignment is calculated based on a metric that measures by “how much” schedulability failed. The HOPA algorithm is based on the distribution of the end-to-end

or global deadlines of each transaction among the different processes that compose that transaction. Once each process is assigned an artificial local deadline, deadline monotonic priorities are assigned in each resource and an analysis of the whole system is carried out. As a result of the analysis, new intermediate deadlines are calculated. The iteration proceeds until a suitable solution is found or some stopping criterion is reached.

Saksena and Hong [Saksena96] proposed a deadline distribution technique based on a critical scaling factor that is applied to the process execution times. The end-to-end deadline is expressed as a set of local deadline assignment constraints. Given a set of local deadline assignments, they calculated the largest value of the scaling factor that still makes the processes scheduable. The local deadline assignment is then chosen to maximize the largest value of the scaling factor.

Jonsson and Shin [Jonsson97] presented a heuristic approach that performs deadline distribution prior to process assignment. The deadline distribution problem is presented in the context of large distributed hard real-time systems with relaxed locality constraints, where schedulability analysis must be performed off-line, and only a subset of the processes are constrained by predetermined assignments to specific processors. The strategy used for deadline distribution uses the concept of critical path. For each transaction graph in the system, first it determines the critical path that minimizes the maximum transaction lateness, then distribute the end-to-end deadline over the processes in the critical path.

Chapter 3: Design Specification by Scenarios and SDL

The increasing complexity of parallel and distributed systems make the development of these systems a complex and labor intensive process. To facilitate and speed up the development process, appropriate and effective techniques must be used during all phases of development life cycle. These should include methods and tools that support the development process from requirement analysis to coding.

Standardized formal description techniques like LOTOS (ISO 8807), Estelle (ISO 9074), and SDL (ITU Z.100) provide a unifying theoretical basis for the construction of dedicated CASE tools that support all phases in the software development and life cycle, namely requirement analysis, design, specification, implementation, test and monitoring of the real system. Modern object-oriented specification languages like SDL'92 supports the software engineering process from object-oriented design down to the generation of executable code. Formalization provides a model which aim is to understand the problem independently of a particular implementation. This model includes the information that is meaningful from a real-world perspective and it represents the external view of the system. The model abstracts from design details in order to give an overview of the system, to postpone implementation decisions, and to allow all valid implementations. It makes use of neither design nor implementation concepts; it defines rather a model that represents the significant properties and functionality of the system.

Formal description techniques describe the structure and the behavior of a system. Their

purpose is to produce correct and unambiguous descriptions, that should ultimately produce efficient implementations. Writing a formal specification consists in understanding and modeling the system and the domain within which it operates. It includes abstracting important real-world features, focusing on what needs to be done, independently of how it is done. The input to the formal specification is a set of requirement statements and a conceptual overview of the proposed system. The output is a formal representation that captures the functional description of the proposed system.

There are many formal methods available nowadays, however, the most common ones are those standardized: SDL (ITU), LOTOS (ISO) and Estelle (ISO). LOTOS [ISO87a] is based on process algebra, both Estelle [ISO87b] and SDL [ITU93a] are based on extended finite state machines.

The methodology and framework we propose in this thesis, for performance modeling and optimization, is generic enough to be applied to any of the standard specification languages. It does not depend on the use of SDL. It does, however, require that the software design description technique used describes the dynamic behavior of the system. This means that the design method shall have concepts for modeling parts of the system and their communication. In this thesis, we focus on SDL and provide further steps of the methodology according to the features of SDL. SDL was selected as an intermediate specification and design language because it is a well established industry standard (SDL-Z.100 and MSC-Z.120). It is specially aimed at the development of telecommunication systems which is the main concern of this research. SDL has evolved to the point where commercial CASE tools are available for graphical editing, simulation, automatic translation to C and C++, and high performance code generation for established real-time kernels.

The rest of this chapter is organized as follows. Section 3.1 gives an introduction to SDL,

which briefly addresses the basic concepts of SDL and the SDL queueing mechanism. Section 3.2 presents the process of design specification using scenarios and SDL/SDP, as well as the type and structure of SDL models that are supported by the proposed performance engineering framework.

3.1 Introduction to SDL

The Specification and Description Language (SDL), was first defined in 1976, then it has been standardized by ITU in 1993 [ITU93a]. The latest version of SDL, SDL'92 supports the software engineering process from object-oriented design down to the generation of executable code. In conjunction with Message Sequence Sharts (MSCs) [ITU93b], system simulation and testing is supported too.

SDL is intended to be well-suited for all systems whose behavior can be effectively modeled by Extended Finite State Machines (EFSM) and where the focus is to be placed especially on interaction aspects. In conjunction with tools, SDL is used by many companies in the telecommunication industry, mainly to design communications software. In addition, it has been employed for the design of real-time and safety critical systems.

There are two main providers of commercial tools for SDL, namely Telelogic with SDT [Telelogic96] and Verilog with GEODE [Verilog96]. The tools support formal specification, validation, simulation, code generation and testing.

Because of the complexity of SDL only an overview of the most important language constructs is presented. For a detailed discussion of all SDL features and language definition, the ITU-T Recommendation Z.100 [ITU93a], or text books on SDL like [Saracco89, Braek93, Oslen94] should be consulted.

3.1.1 An SDL system and its structure

An SDL specification describes an abstract machine which is called an SDL system, and composed of a set of extended finite state machines (EFSMs), called processes. An SDL system communicates with its environment by exchanging signals via communication links called channels, as shown in Figure 3.1. The channels form the logical interface to the system environment. From the viewpoint of the system, the objects in the environment behave like processes. Inside the SDL system, the EFSMs communicate asynchronously via communication links by exchanging signals.

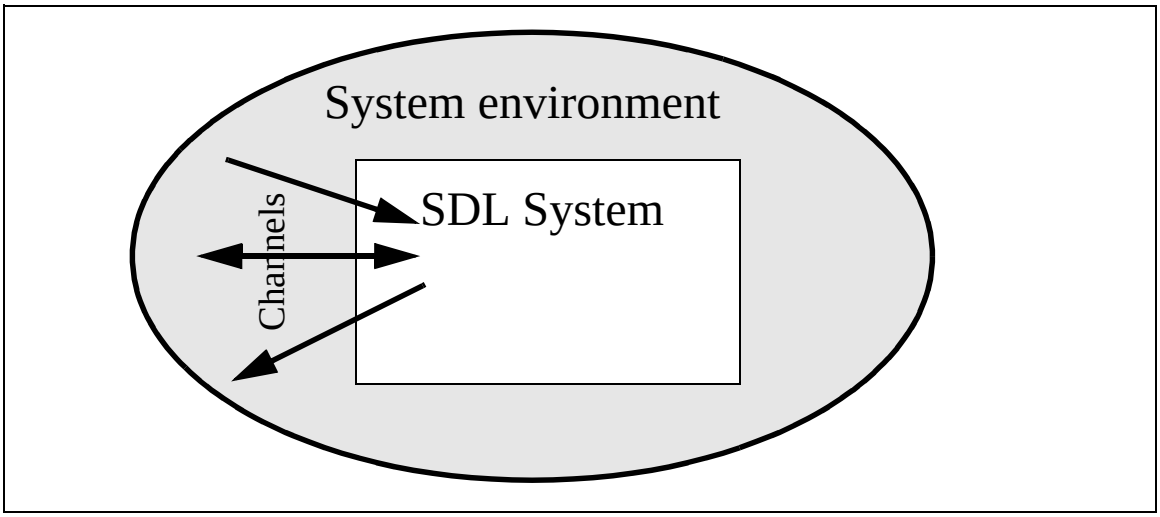


Figure 3.1 An SDL system

SDL specifications are fully hierarchically structured as a tree. The root of the tree refers to the SDL system specification which typically consists of a set of blocks. Blocks themselves can be refined by other blocks or by SDL processes. However, each leaf of the tree has to be an SDL process. The communication structure between SDL processes is static and all potential communication channels have to be given in the SDL specification.

3.1.2 The SDL process

An SDL process is defined by an Extended Finite State Machine and defines the dynamic behavior of a system.

- A Finite State Machine is an abstraction for the control of a process, with a limited number of states and can perform transitions between those states.
- Communicating FSMs exchange information via stimulus and response. The receipt of a stimulus causes an FSM to perform a transition possibly resulting in the transmission of a response.
- Extended FSMs include variables which record secondary state information that can be used in a transition to decide the next state, i.e. an EFSM is a finite state machine (FSM) which additionally to its states can hold and manipulate data.

According to the EFSM model, an SDL process is either in a state transition or is waiting for an event which triggers the next state transition. The state defines what actions a process is allowed to take, which events it expects to happen, and how it will respond to those events. A process remains in a state until an event, such as signal instance, that is handled by that state occurs; if a signal instance is not handled in that state a null transition back to the same state is performed (i.e. the signal is discarded). Once a signal instance arrives that can be handled, the transition actions are performed and the process enters its next state. During a state transition a process can manipulate data, set and reset timers, call procedures and send signals to other processes or to system environment. A state transition can be triggered by a signal from another process, by a timer signal, or by a specific data value. Additionally, SDL allows to specify spontaneous transitions (spontaneous transitions allow to describe nondeterminism on process level).

A process can contain many different states, to allow the process to perform different actions

when a signal is received. A process can be in only one state at a time and a signal handled by a state can have only one transition. Processes can only disappear from an SDL system through self-termination.

A process definition represents the specification of a type of a process; several instances of the same type may be created and exist at the same time; they can execute independently and concurrently. Each process instance has its own unbounded FIFO input queue, which is initially empty. An input signal is placed in the input queue of a process instance as it arrives. The input queue can retain any number of input signals ordered according to their arrival time. Signals in the input queue can be discarded, consumed or saved. Signals are discarded, consumed or saved by a process instance only when it is waiting in a state.

3.1.2.1 Signals and signal queue of an SDL process.

In SDL, inputs and outputs are related to signals which are exchanged between the processes of an SDL system. For communication purposes an infinite FIFO signal queue is associated with every process. All signals arriving at an SDL process are put into its signal queue. If in a given state the signal queue is not empty, the first signal (in FIFO order) is removed from the queue. It is checked whether this signal can initiate a state transition. If yes, the transition is performed (the signal is consumed); if no, the signal is discarded.

3.1.2.2 Process Wake Up

SDL has four constructs that introduce the transitions.

Input signal. An input symbol attached to a state means that if the signal named within the input symbol arrives while the process is waiting in this state the transition which follow the input symbol should be interpreted. When a signal has triggered the interpretation of a transition, the

signal no longer exists and is said to have been consumed.

Input with enabling condition. Enabling conditions allow conditional reception of signals based on the specified enabling condition. If the condition is true, the signal is consumed and the transition is interpreted. If the condition is false, the signal is saved and the process remains in the state until either another signal arrives or until the condition changes from being false to being true

Continuous Signal. Represents a transition selected by a boolean expression only when there are no signal instances to consume (i.e there can be no signal instance consumption). If there are no consumable signals then if the boolean expression is True, the transition will be selected.

Saves. Sometimes a signal should not be discarded even if it can not be consumed next by the process (due to the process state). Instead, the signal should be saved for future use, after the following signals have been processed. In SDL this can be achieved by means of the save construct. The save concept allows the consumption of a signal to be delayed until one or more other signals, which arrive subsequently, have been consumed. Unless the save concept is used, signals are consumed in the order in which they arrive. The concept of save can be used to simplify processes in cases where the relative arrival order of some signals are not important and the actual arrival order is indeterminate. The save construct introduces a second signal queue for a process. This second queue is prioritized over the normal signal queue. For a state transition a process will first look into the prioritized queue. Only if this queue is empty or if all the remaining signals already have been saved in the current state, the signals in the normal queue will be consumed.

An example that demonstrates the save concept is presented in Figure 3.2. In this example, it is assumed that the finite state machine is in state S and the input queue has the three signals

Z, W and X. In each state, the queue is traversed from front to back. The sequence of actions performed by this machine is summarized in Figure 3.3.

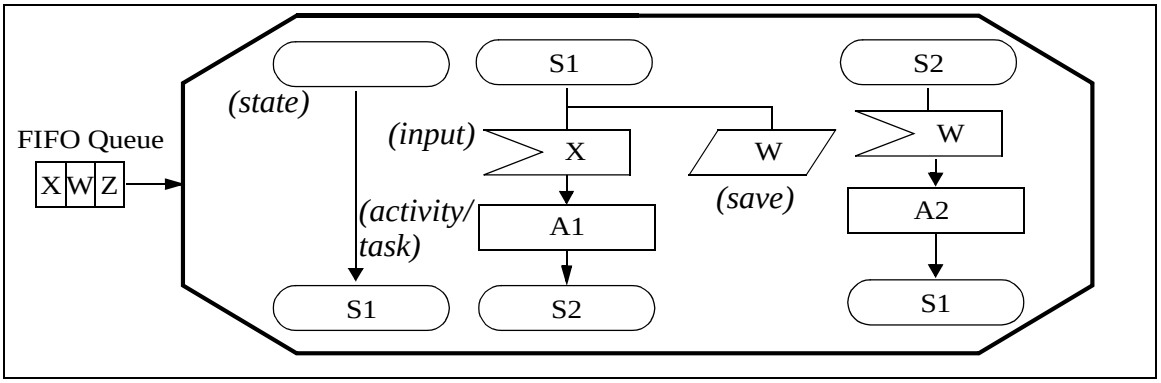


Figure 3.2 An example on the Save concept

Curr. state	Event
S1	The first signal in queue, Z, will be discarded
S1	The first signal in queue, W, appears in a save symbol and remains in queue.
S1	The second signal, X, will be consumed (explicit input), action A1 will be executed and state S2 will be entered
S2	The saved signal W will be consumed, action A2 will be executed and state S1 will be entered

Figure 3.3 The sequence of actions performed by the state machine in Figure 3.2

3.1.2.3 Internal Actions

Activities. An activity is used in a transition to represent operations on variables or to represent special operations by means of informal text. In the context of SDL, activities are called tasks.

The decisions. SDL offers the decision construct to specify value driven state transitions.

Outputs. An output is the sending of a signal from one process instance to another (or to itself).

Because control over the consumption of the signal is associated with the receiving process the semantics directly relating to the output is relatively simple. From the point of view of the sending process an output can often be regarded as an instantaneous action which, once completed, has no further direct effect on the sending process, which will not be directly aware of the fate of the signal.

Create a process. Processes (i.e. process instances) can be created as a result of an explicit request or at the system creation. The explicit create request may be issued only by another process in the same block of the process being created and allows the specification of actual parameters for transferring information to the new instance created.

Timer instructions (set/reset). The need to measure time and request time-outs in a system is met by timers and a set of operations performed upon them. In the SDL model “a timer” is a meta-process (object) which is associated with a process. A timer can be active or inactive. If it is active, after a predefined time it will put a timer signal into the signal queue of the process. The timer signal can be consumed and trigger state transition like any other signal. A timer has to be declared in an SDL process definition. During a state transition a timer can be set and reset by means of “set” and “reset” instructions which are written in “activity” symbols. The set operation requests a time-out to occur at a specified time, and the reset operation cancels the specified time-out. For defining a point of time in future, the expression “now” can be used, now always denotes the current time at which the set instruction is evaluated. By means of a now it is possible to define the expiration of a timer relative to the corresponding set instruction. After expiration, the timer produces a timer signal, which has the same name as that of the timer, and puts it into the signal queue of the associated process. Once a timer has been set, it can be canceled with the reset instruction. In this case the process behaves as if the timer never had been set. Any timer signal still located in the signal queue will be deleted.

3.1.2.4 The declaration and manipulation of data

The base model of an SDL process is an extended finite state machine which can hold and manage data. The data is local to the process and exists in the form of values and variables. The variable and its type must be declared. During a state transition an SDL process can manipulate its local data. Data manipulation is specified within an “activity” symbol. An activity can only include assignment statements.

3.1.3 Communication in SDL

In SDL communication is based on three sets of constructs:

- explicit signal communication (sending of a signal to receiver).
- communication via shared data (sharing of information).
- communication via instance activation (creation of a receiver).

3.1.3.1 Explicit Signal Communication.

The SDL processes communicate by exchanging signals. For this purpose each SDL process has its own infinite FIFO signal queue. The signal queues and the corresponding SDL processes operate independently and in parallel. Processes which want to communicate with a certain partner process have to put their signals into the queue of the partner. A sender may produce infinitely many signals without waiting for the receiver to consume them. If two signals reach a process simultaneously, they are buffered in random sequence.

Communication takes place via communication links which in SDL are called signal routes and channels. The channels are typed; meaning that they can only convey signals of certain types.

3.1.3.2 Communication via shared data

Any information in SDL is always owned by a certain process instance and this instance is the only one that has the right to change the value of the information. However the process definition of a process instance may grant access right to the environment of the instance so that the information can be visible to other process instances (of the same or other process definitions). The sharing of information is obtained by using the pairs of constructs. REVEALED-VIEW and EXPORT-IMPORT. The first pair can only be used within a block boundary, the later does not impose any limitation.

In the Reveal/View communication the senders declares that an information associated with a variable is disclosed to other viewers. Once an information is disclosed through a REVEALED, any process instance within that block definition can access it. The access to the information is instantaneous, and does not take any time. More viewers can view the same information concurrently. The competition between the owner changing the information and viewers is solved by the assumption that the change of an information takes no perceivable time.

In the Export/Impor**t** communication, a process can declare one or more of its variables as “exportable” which has the effect that all other processes (no matter the block they belong) can import a copy of the value of the variable upon request. Any subsequent change of information is hidden from the importers that continue to access the exported value. In this way there is no concurrency problem between the exporter and the importers. The import of information may involve time, even though this time is assumed to be negligible from a specification viewpoint.

3.1.3.3 Creation of a receiver

Communication may also occur through the creation of the receiver by the sender. The

creation takes no time and therefore the information associated with the create clause is immediately available to the receiver. In this way of communication a synchronization between the creating process instance and the created process instance is established. However, immediately after the creation the two instances (the creating and the created one) continue their processing asynchronously.

The creation can only occur within a block boundary, therefore this way of communication can only occur within a block boundary. Once a receiver exists the only possible forms of communication are via signals and through shared variables.

3.1.4 Signals and data

The communication between different processes, and the communication between processes and the environment is realized by means of signals. Signals can be defined at system level, block level, or in the internal part of process definition. Signals defined at system level represent signals interchanged with the environment and between system blocks. Signals defined at block level represent signals interchanged between processes of the same block. Signals defined within a process definition can be interchanged between instances of the same process type or between services in the process.

Signals can also be used for transmitting data from one process to another. In this case a signal has one or several assigned values which are called parameters. The parameters are assigned to a signal in an output symbol. When the signal is consumed, its parameters can be assigned to variables for further processing. This is specified in a corresponding input symbol. The types of the transmitted parameters must correspond with the types of variables in the input symbol at the responder.

Signals are sent along signal routes between processes and on channels between blocks or when interchanged with the environment. Signals traveling along a communication channel can experience a nondeterministic delay. Once inside a block, a signal is routed immediately to the appropriate process instances. Once at a process instance the signal is put in a FIFO queue for that instance, and held there until it is consumed by the instance.

Channels. Channels are the communication media between different blocks of the system or between blocks and environment. SDL Channels have FIFO semantics, i.e. a signal will never overtake a signal which is previously sent. They can be unidirectional or bi-directional. Transmission of signals by channels can experience a nondeterministic delay. This means that a signal A sent before another signal B on a different channel, may arrive later than B at its destination endpoint. This is intended to model the real behavior of communication channels, where re-ordering occurs with finite probability. Furthermore, an SDL channel is safe, i.e. no signal will get lost or be modified during transmission. SDL allows to specify whether a channel may delay the signal transmission or not.

Signal Routes. Represents the routing of signals from communication channels to processes or between processes in the same block. Signals presented to the origination endpoint are delivered to the destination endpoint in the same order with no delay. This reflects the fact that signal routes are only used to specify routing and not transmission along communication channels. Signalroutes can be unidirectional or bidirectional. They are very similar to channels, except that a signal route can not delay the transmission of signals.

3.1.5 The SDL Queueing Mechanism

At every state, every signal is treated in one of the following ways: (i) it is shown as an input, (2) it is shown as a save, as was discussed in Section 3.1.2.2, or (3) it is covered by an implicit

input leading to an implicit null transition, that is the signal is consumed without any action being performed. It is discarded.

On arrival, signals enter the queue and when the process reaches a state, the signals in the queue are reviewed one at a time in the order in which they arrived. A signal covered by an explicit, or implicit, input is consumed and the related transition executed. A signal shown in a save is not consumed and remains in the queue in the same sequential position and the next signal in the queue is considered. No transition follows a save. An example of how the (logical) queueing mechanism of the process works is shown in Figures 3.4 and 3.5 [Wohlin91].

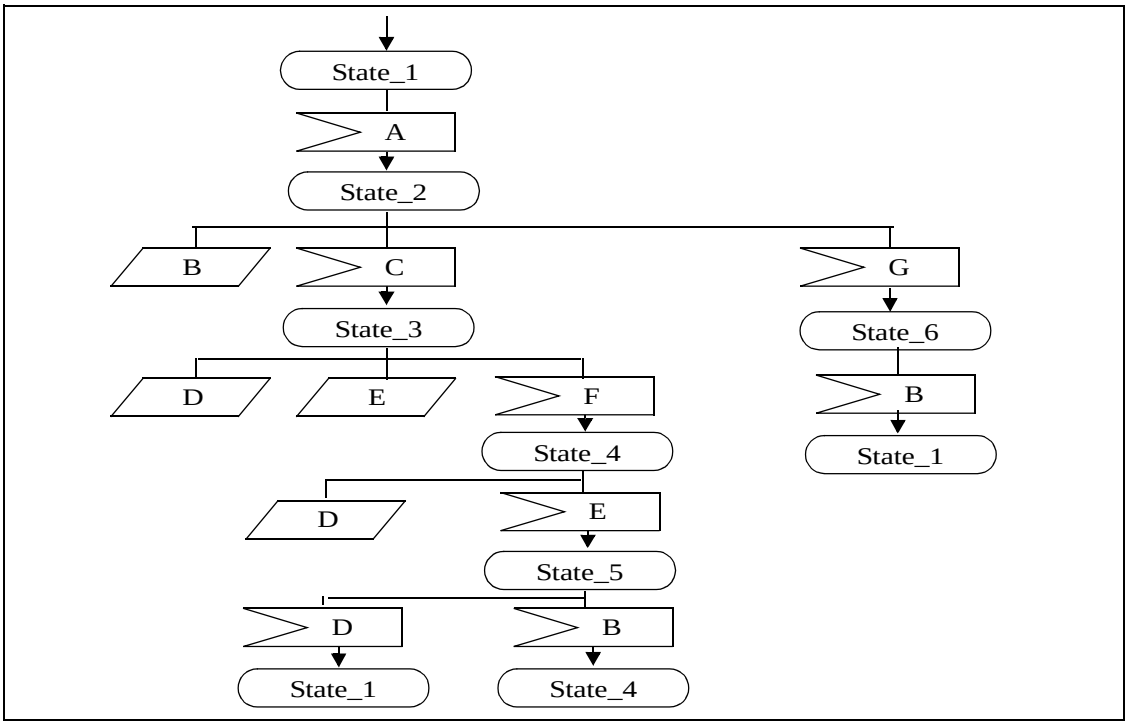


Figure 3.4 Example of a complex SDL diagram with numerous inputs and saves

Curr. state	Event	Queueing
State_1	(Process arrives at State_1 with signals A,B,C,D and E in queue) The first signal in queue, A, is consumed and transition to State_2 triggered.	order of arrival → A B C D E F
State_2	The first signal in queue, B, appears in a save symbol and remains in queue.	
State_2	The second signal,C, is consumed (explicit input) and transition to State_3 is triggered.	
State_3	The first signal in queue, B, is consumed (implicit input).	
	Signal F arrives and enters queue.	
State_3	(On reaching State_3 again) the first signal in queue,D, appears in a save symbol and remains in queue.	
State_3	The second signal, E, appears in save symbol and remains in queue.	
State_3	The third signal, F, is consumed (explicit input) and transition to state_4 is triggered.	
State_4	The first signal in queue, D, appears in a save symbol and remains in queue.	
State_4	The second signal, E, is consumed (explicit input) and transition to state 5 is triggered.	
State_5	The first (and only) signal in queue,D, is consumed (explicit input) and transition to state_1 is triggered.	

Figure 3.5 Operation of queueing mechanism in the system of Figure 3.4

3.2 Design Specification by Scenarios and the Scenario Design Paradigm (SDP)

Our designs are derived from scenarios which can be expressed for the present purposes by Message Sequence Charts (MSCs) showing sequences of activities executed by different processes. An example of three scenarios is shown in Figure 3.6. The first scenario, illustrated in Figure 3.6(a), represents a request-response interaction pattern between the environment (ENV) and process A. The second scenario depicts a chain of requests from ENV to process A then to process C. While, the third scenario describes a request-forward-reply interaction pattern between Env, process B and process C.

SDL processes which execute these activities are constructed using a prototype software design technology, developed inside Nortel Networks, called Scenario Design Paradigm (SDP) [Jedrysiak94, Jedrysiak96, Cameron97], and was described in Section 2.1.2. SDP allows designers to express the behavior of a real-time system as a set of concurrent scenarios using a scenario language. Each scenario is a system response to an external event. The scenarios are specifications of the actions the system must take to generate the required outputs given the inputs. The actions are processes sending messages to one another and invoking functions. The active components in the scenarios are messaging entities and the form of their interaction is asynchronous messages.

The scenario language is like a Message Sequence Chart language in textual form but with extended structural syntax that facilitates the translation of the scenario language into SDL. A sample scenario code that implements the MSCs shown in Figure 3.6 is presented in Figure 3.7. The scenario language uses operating system semantics to specify scenarios. The abstractions are system, scenarios, messaging entities and messages. A system is a set of scenarios. A

scenario is a sequence of entities sending messages to each other. A scenario always begins with a message reception event. A system response may be contained within one scenario, or it may be expressed by a set of scenarios. A system response always begins with receiving a message from the environment; it may or may not require a message being sent to the environment.

Scenario synthesis is defined as the generation of component state machines from a set of scenarios [Cameron97]. The scenario language compiler performs a kind of state machine synthesis at the process level. The compiler constructs a state machine for each active system component by extracting the behavior of that component from each scenario in which it appears. The state machine language is SDL. In addition, the scenario language compiler generates structural definitions to satisfy the requirements of a compilable SDL specification. The relevant structures are blocks for the containment of processes, channels for the connection of blocks to the system boundary or to each other, and signal routes for the connection of processes to a block boundary or to each other. The scenario language contains syntax for declaring blocks and channels but no syntax for declaring signal routes. In the absence of explicit structural declarations the compiler will generate default block and channel declarations. The defaults are one block per process and one channel per message. A signal route is created for each message.

Each compile produces SDL Graphical Representation (GR) files. These files contain graphical representations of the block architecture as well as the process state machines. Once the system is synthesized into a set of SDL state machines, it can be executed in an SDL CASE tool simulator or on the target product platform. It can be analyzed to verify its behavior.

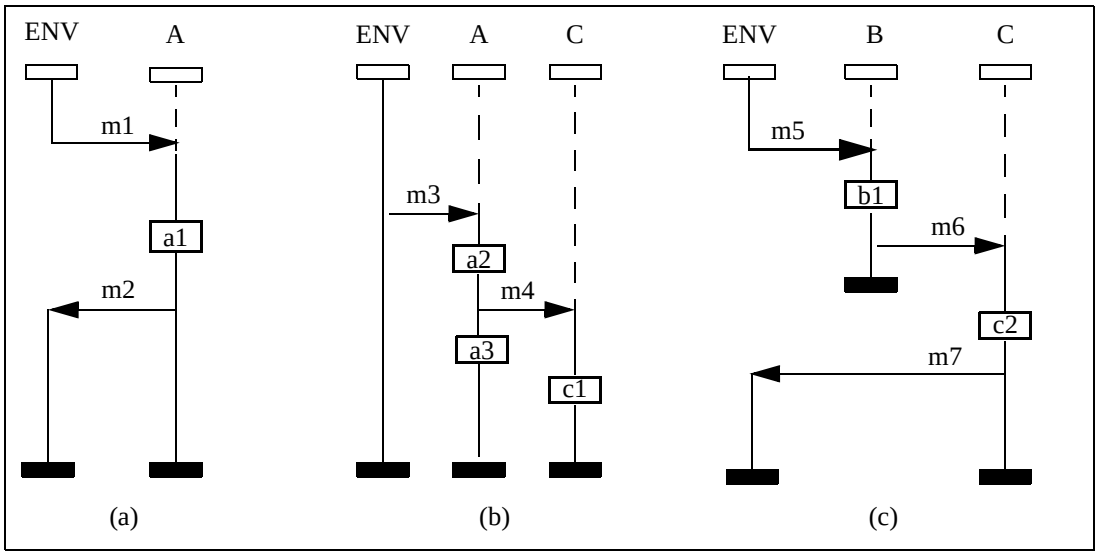


Figure 3.6 An example of three scenarios (a) request-response, (b) chain of requests, (c) request-forward-reply

```
System example

Scenario SysRespOne
  step 1: ENV sends m1 message to processA(a1)
  step 2: processA sends m2 message to ENV
end scenario

scenario SysRespTwo
  step 1: ENV sends m3 message to processA (a2)
  step 2: processA sends m4 message to processC(c1)
  step 3: processA(a3)
end scenario

scenario SysRespThree
  step 1: ENV sends m5 message to processB (b1)
  step 2: processB sends m6 message to processC (c2)
  step 3: processC sends m7 message to ENV
end scenario

end system
```

Figure 3.7 A sample scenario code that models the MSCs shown in Figure 3.6

3.2.1 SDL models that are supported by the model-builder

The Scenario Design Paradigm approach to design is based on an asynchronous process model. Whenever a process receives a message, it processes the message to completion. Every time an event is received, it is processed by calling the same sequence of transition code. Complexities arising out of combinations of input events and state variables are hidden in the transition code. This is a common design style in telecommunications; any message can be received at any time, and is then processed to completion. Accordingly, in SDP the SDL processes are designed with one state, which follow a simple receive-execute loop, as shown in Figure 3.8. That is, the process receives and handles any message at any time; it never blocks waiting for a particular message. However the process may retain state data about messages it expects to receive, which allow it to resume processing on some partially completed response.

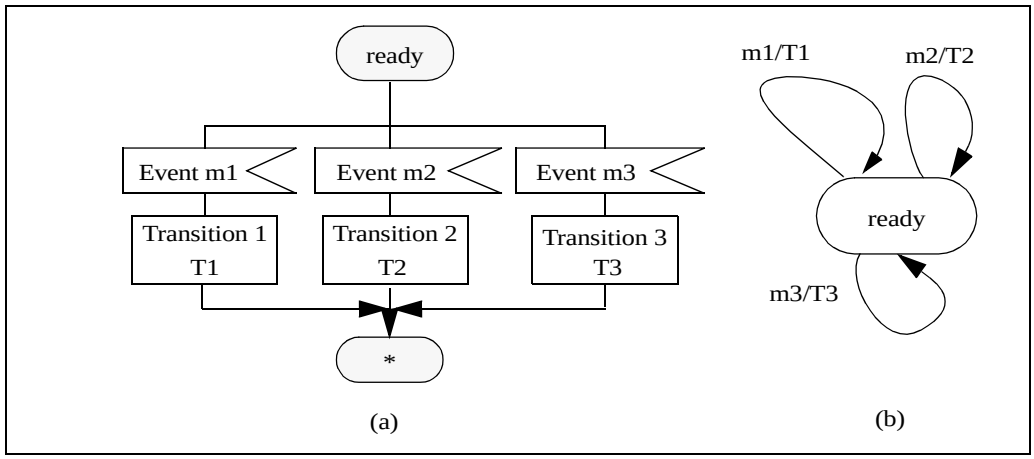


Figure 3.8 A typical model of an asynchronous process. (a) SDL diagram. (b) conventional event/action form

The Scenario Language has constructs for specifying states and state changes. Accordingly, SDP can synthesize SDL processes with general finite state machines. However, SDP does not have constructs to support the save mechanism. So, it does not guarantee the successful

consumption of all messages exchanged in the system. Because of the lack of support of the save concept, some messages may be unintentionally discarded at some states.

The current version of the model builder supports both styles of system design in principle. It can model systems designed with the asynchronous process model as well as systems designed with the general process model that can have many states but do not use the save concept. More general state-machine structure also can be modeled, but would require additional machinery for interpreting the SDL traces. This research has concentrated on designs that use the asynchronous process model.

3.2.2 Specifying models with join and fork-join patterns

The scenario language semantics, in its present form, includes forking of a thread, but not joining. For this research, we wanted to include systems with parallel parts including joins, so the scenario language was extended, with the aid of some functions written in the C language, to implement (simulate) the semantics of the join events. An example implementation of these functions is shown in Figure 3.9. The goal of these functions is to log, during run time, some tags in the simulation traces that allow the model-builder to identify the threads that need to be joined and the location of the join events. These functions can be referenced in the scenario code using the procedure call syntax.

The purpose of the function “init_conditions()” is to initialize the status of all conditions involved in all “join” events in the model to FALSE, i.e. to initially indicate that all conditions has not been satisfied yet. The purpose of the function “set_condition(cond)” is to set the status of the condition specified by the argument “cond” to TRUE, i.e to indicate that the specified condition has been completed. The function “checkJoin(cond1, cond2, &status)” returns TRUE (i.e. sets its status argument to TRUE) if both conditions specified in its argument list are found

to be completed, otherwise it returns FALSE.

The purpose of the function printGotoJoin(join_name) is to record a message “GOTO_JOIN <join_name>” in the simulator output indicating that the current thread will go to wait at the specified join point. While the purpose of the function printEndJoin(join_name) is to record a message “END_JOIN <join_name>” in the simulator output indicating the completion of the last thread of the specified join point. Examples of scenario codes that implement (model) several patterns that fork-and-join [Franks97] are depicted in Figure 3.10 through Figure 3.11

Using these functions, the proposed structure and semantic of a process that has multiple threads running in parallel which need to be joined, before the process can resume its operation, is described as follows. Whenever a thread ends, it first declares its completion using the function “set_condition(cond_name)”. Then using the function “checkJoin <join_name>”, it checks the completion status of the other threads which are involved in the same join event. If the thread finds that all conditions of the joint events have been satisfied, it records the message “END_JOIN <join_name>”, and continues executing the activities after the join point. Otherwise, it records the message “GOTO_JOIN <join_name>” and blocks.

Note that the current implementation of the checkpoint function assumes that join point has only two incoming threads (branches). To model join points with more than two incoming arcs, new extended versions of the checkjoin function need to be added to the library.

```
/* join_lib.c  */

#include <scttypes.h>

#define TRUE 1
#define FALSE 0

/* declare all conditions that are involved in join events */
int m1_rcvd, m2_rcvd, a1_completed, a2_completed ;

/* initialize all conditions that are involved in join events to FALSE */
int init_conditions(void)
{
    m1_rcvd = FALSE;
    m2_rcvd = FALSE;
    a1_completed = FALSE;
    a2_completed = FALSE;
    return(0);
}
int set_condition(int *cond)
{
    *cond = TRUE ;
    return(0);
}

int checkjoin(int cond1, int cond2, int *status)
{
    *status = FALSE;
    if ((cond1==TRUE) && (cond2==TRUE) ) *status=1;
    return(0);
}

int printGotoJoin(char *s)
{
    char string[100];
    sprintf(string, "**** GOTO_JOIN %s\n", s);
    xPrintString(string);
}


int printEndJoin(char *s)
{
    char string[100];
    sprintf(string, "**** END_JOIN %s\n", s);
    xPrintString(string);
}
```

Figure 3.9 A library of C functions that supports modeling the "join" events

3.2.3 Examples of scenario codes that describe fork-join patterns

There are two forms of fork-join behavior which can occur within an application. The two forms are based on whether the fork and join take place within the same process, or in two separate processes [Franks97].

1) **Inter-Process Fork-Join** occurs when messages originate from a common client process, follow independent routes, then join at a common server process.
2) **Intra-Process Fork-Join** occurs when the fork and join take place within the same process.

Figure 3.10 (a) shows an example of inter-process fork-join behavior expressed by MSC, and Figure 3.10 (b) shows the scenario language code which models that behavior. The scenario code sets up a system that does the following.

- When process t1 receives message m1 from the environment, it responds by executing activity a then sends message m3 to process t3.
- When process t2 receives message m2 from the environment, it responds by executing activity b then sends message m4 to process t3.
- When process t3 receives message m3, it will do the following actions.
  i) execute activity c
  ii) declare that activity c has been completed, by setting its associated completion flag to TRUE.
  iii) check the completion status of activity d. If process t3 finds that activity d has been completed, it will call the function “printEndJoin”, execute activity g then send message m5 to environment. Otherwise, it will call the function “printGoToJoin” and block.
- Similarly, when process t3 receives message m4, it will do the following actions.

i) execute activity d

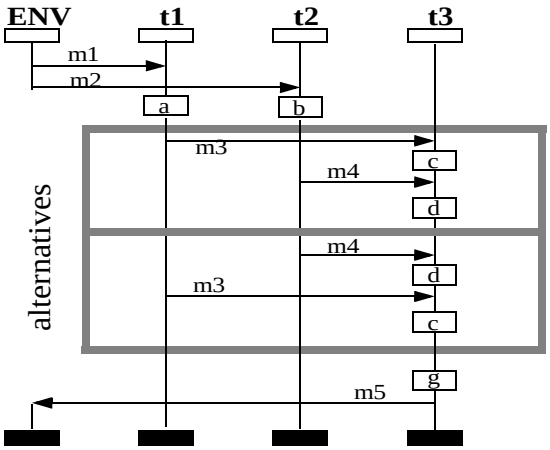
ii) declare that activity d has been completed, by setting its associated completion flag to TRUE.

iii) check the completion status of activity c. If process t3 finds that activity c has been completed, it will call the function “printEndJoin”, execute activity g then send message m5 to environment. Otherwise, it will call the function “printGoToJoin” and block.

Figure 3.11 (a) shows an example of intra-process fork-join behavior expressed by MSC, and Figure 3.11 (b) shows the scenario language code which models that behavior. The scenario code sets up a system that does the following.

- When process t1 receives message m1 from the environment, it responds by performing the following activities:
  i) execute activity a
  ii) sends message m2 to process t2
  iii) execute activity b
  iv) declare that activity b has been completed, by setting its associated completion flag to TRUE.
  v) check the reception status of message m3. If process t1 finds that message m3 has been received, it will call the function “printEndJoin”, execute activity d then send message m4 to environment. Otherwise, it will call the function “printGoToJoin” and block.
- When process t2 receives message m2, it responds by executing activity c then sends message m3 to process t1.

- When process t1 receives message m3 from the process t2, it responds by performing the following activities:
  i) declare that message m3 has been received, by setting its associated reception flag to TRUE.
  ii) check the completion status of activity b. If process t1 finds that activity b has been completed, it will call the function “printEndJoin”, execute activity d then send message m4 to environment. Otherwise, it will call the function “printGoToJoin” and block.

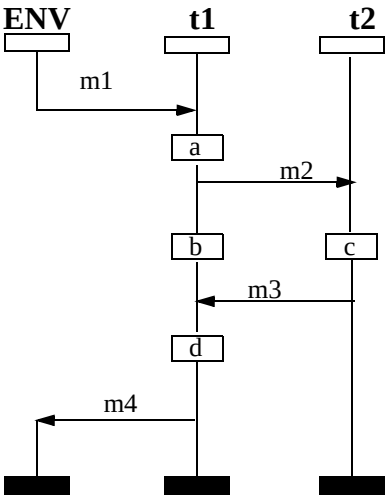


(a) MSC diagram

```
system Inter_process_Fork_Join

library
   join_lib
endlibrary

scenario s1
  step 1: ENV sends m1 to t1(a)
  step 2: ENV sends m2 to t2(b)
  step 3: t1 sends m3 message to t3
  step 4: t2 sends m4 message to t3
end scenario

scenario s2
  step 1: message m3 received
            t3 (c)
            t3(set_condition(&c_completed))
            t3(checkJoin(c_completed,d_completed,&status))
  step 2: if (status == 1) then
                t3(printEndJoin("join1"))
                t3(g) sends m5 message to ENV
          endif
  step 3: if (status == 0) then
                t3(printGotoJoin("join1"))
            endif
end scenario

scenario s3
  step 1: message m4 received
            t3 (d)
            t3(set_condition(&d_completed))
            t3(checkJoin(c_completed,d_completed,&status))
  step 2: if (status == 1) then
                t3(printEndJoin("join1"))
                t3(g) sends m5 message to ENV
            endif
  step 3: if (status == 0) then
                t3(printGotoJoin("join1"))
            endif
end scenario
end system
```

(b) Scenario code

Figure 3.10 An example scenario code of Inter_Process Fork_Join



(a) MSC diagram

```
system Intra_process_Fork_join

library
   join_lib
endlibrary

scenario s1
  step 1:  ENV sends m1 to t1(a)
  step 2:  t1 sends m2 message to t2(c)
  step 3:  t1(b)
  step 4:  t1(set_condition(&b_completed))
            t1(checkJoin(m3_received,b_completed,&status))
  step 5:   if (status == 1) then
               t1(printEndJoin("join1"))
               t1(d) sends m4 message to ENV
            endif
  step 6:  if (status == 0) then
               t1(printGotoJoin("join1"))
            endif
 end scenario

scenario s2
  step 1:  message m3 received
            t1(set_condition(&m3_rcvd))
            t1(checkJoin(m3_received,b_completed,&status))
  step 2:  if (status == 1) then
               t1(printEndJoin("join1"))
               t1(d) sends m4 message to ENV
            endif
  step 3:  if (status == 0) then
                t3(printGotoJoin("join1"))
            endif
end scenario

scenario s3
   step1:  message m2 received
            t2 sends m3 message to t1
end scenario
end system
```

(b) Scenario code

Figure 3.11 An example scenario code of Intra-Process Fork-Join

Chapter 4: The Model Builder

4.1 Introduction

This chapter describes an approach for constructing LQN performance models automatically from design specifications which are defined in a standard design environment based on SDL. The software design is assumed to be expressed in an executable SDL model, and to conform to a certain asynchronous style. The designer must supply scenarios [Jedrysiak94,96] which are used to drive the design model in an execution mode to produce traces, as well as additional information in the form of resource functions and the configurations to be evaluated as described later. The traces are interpreted to extract the structure and some of the parameters of a performance model, in a way extended from that described for “angio traces” in [Hrischuk95].

The procedure used in this work is outlined in Figure 4.1. To build an LQN model from traces, the traces are post-processed into “angio traces” which describe the history of the trace in a structured fashion, and then into performance sub-models (one per trace). These, the resource functions, and some necessary configuration data (such as processors or nodes, task allocation, task priorities and rates of events) are finally merged into the model and used to make evaluations. Figure 4.2 shows an example with submodels (a), (b) and (c) created from scenarios (a), (b) and (c) of Figure 3.6, and the final model that combines them is shown in Figure 4.3.

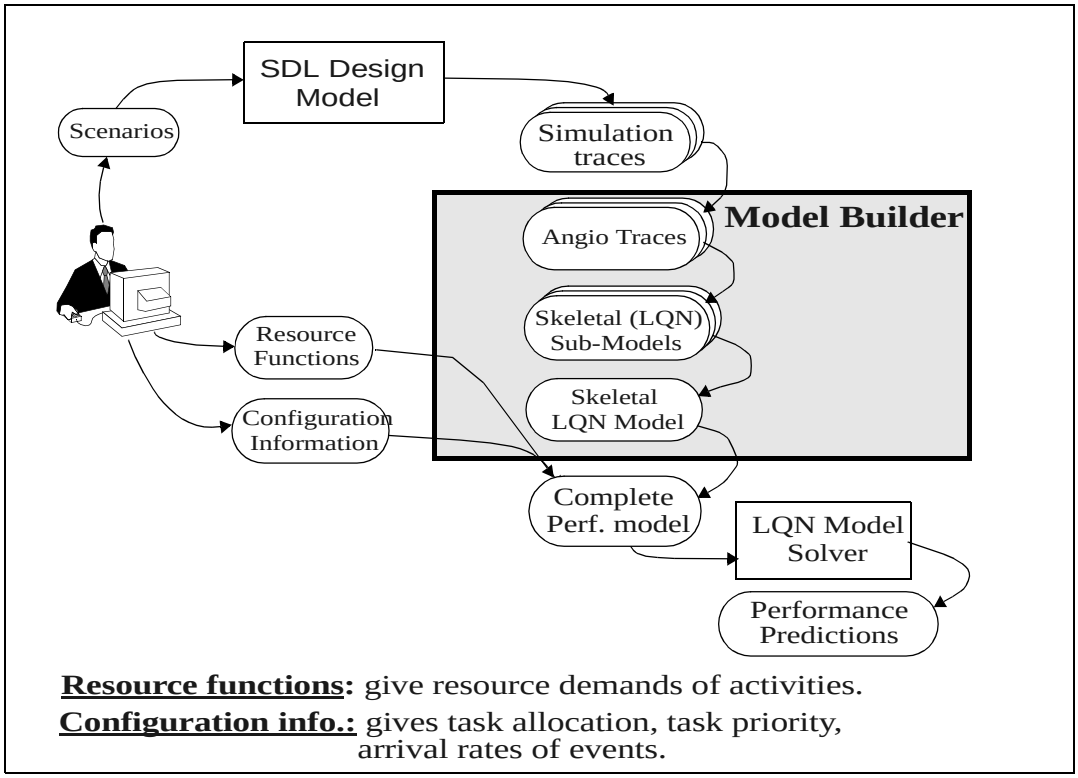


Figure 4.1 The Model-Builder

The approach is implemented in a tool called "The Model Builder", using Perl. The Model Builder takes as input:

1) traces of the representative scenarios of the design.
2) resource demands of activities.
3) environment (configuration) information which is not in SDL specification such as: process allocation, priorities, and workload parameters (traffic load).

The output of the tool is an LQN performance model, of the design that has been traced.

The rest of this chapter is organized as follows. Section 4.2 describes the process for defining scenarios and collecting their SDT traces. Section 4.3 describes the process for extracting LQN-submodels from traces. The process of merging the submodels and completing the LQN model

is described in Section 4.4. Finally, the algorithms used to implement the approach are presented in Section 4.5.

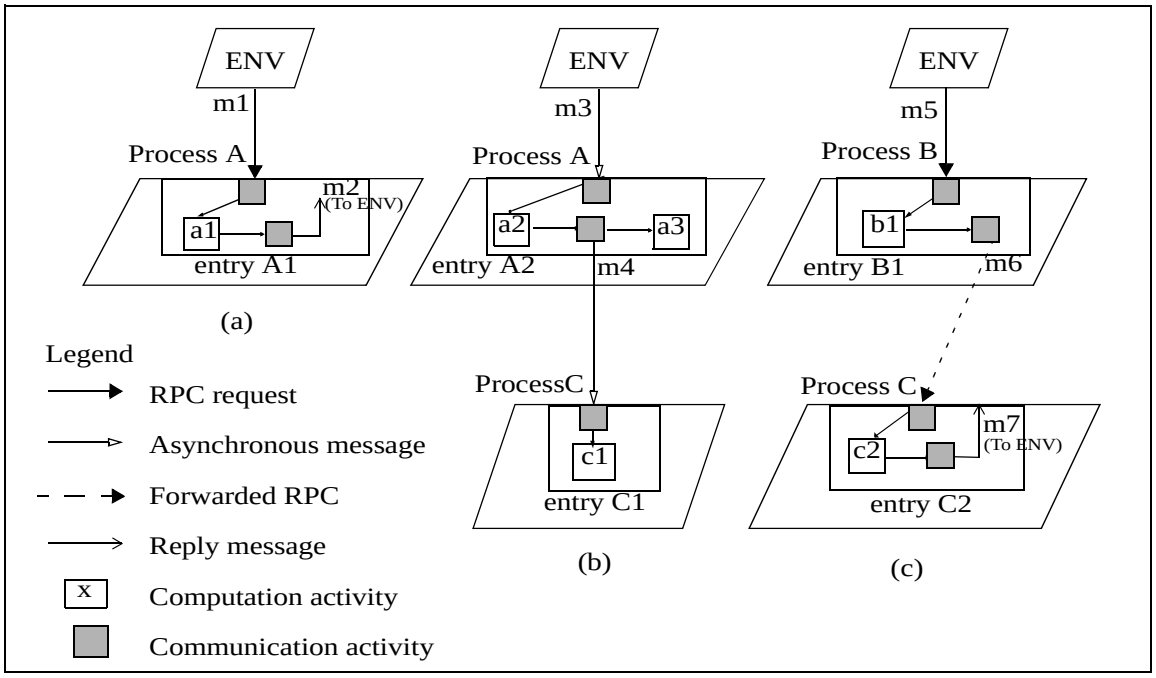


Figure 4.2 An example of three LQN submodels for the interactions shown in Figure 3.6

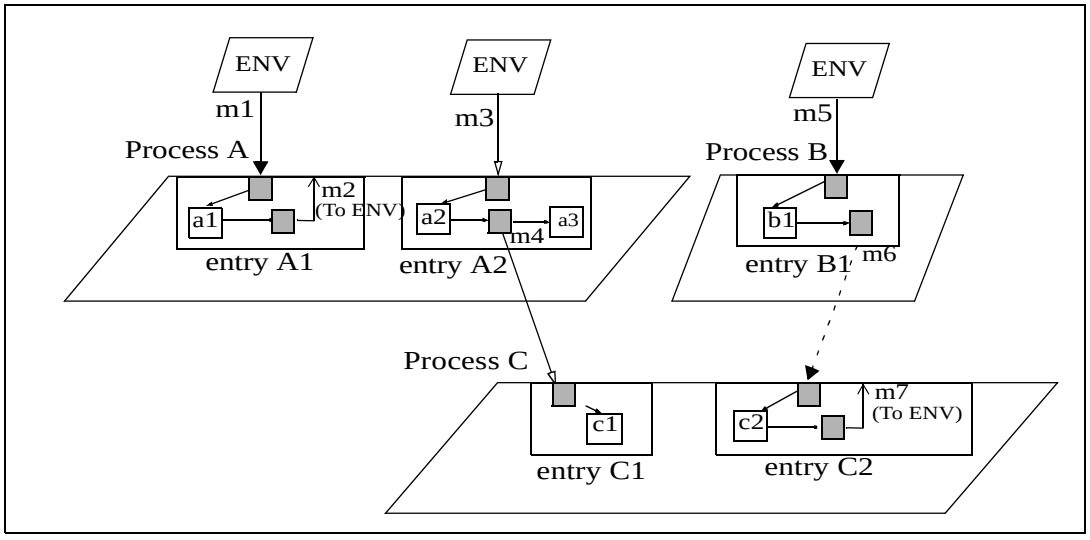


Figure 4.3 The merged model

4.2 Define scenarios and capture SDT traces

The design evaluation process described here is based on scenarios, as was discussed in Section 3.2. Scenarios are sequences of activities (actions) that specify the desired behavior of a software system. They are slices of the system behavior, which can be considered one at a time in terms of the system model. Each scenario describes the behavior of the system in response to an external event.

In this work, the choice of scenarios relies on the experience and judgment of domain experts. It is possible that some important scenarios may be overlooked, due to the complexity of the system. However, a well defined set of scenarios will reveal enough of the important behavior to give a useful performance model. Some guidelines for choosing scenarios were discussed in [Smith90]. The adequate coverage of behavior by scenarios is a very important issue for model building but beyond the scope of this research.

For model building, the set of scenarios that describe the system is first defined using the Scenario Language [Jedrysiak94] and then compiled to an SDL model using the Scenario Design Paradigm [Jedrysiak94, Cameron97]. The SDL model is then simulated using SDT and its traces are recorded.

To automate and speed up the process of stimulating the scenarios and collecting their SDT traces, an SDT command script *"run_collect_traces"* was developed. The script can be invoked at the command line as follows.

model.sct < run_collect_traces

Where, model.sct is the name of the executable version of the SDL design (which could be

generated automatically from the scenario specifications using the scenario compiler).

The script “run_collect_traces” first initializes the system then injects the triggering stimuli to the system. Each triggering stimulus initiates the execution of one scenario. Every time the script injects a stimulus into the system, it simulates the system and records the traces of the simulation into a special file called “model.traces”. The commands of the script are listed below.

```
output-v init -          //sends the initialization message to the system.
go                       //simulate (initialize) the system
set-trace 6              //set simulator to record all events
log-on model.traces      //set simulator to record traces in model.traces
output-v stim1 -         //inject message stim1 to the system
go                       // simulate the system
output-v stim2 -         //inject message stim2 to the system
go                       // simulate the system
quit
```

4.3 Building Skeletal LQN submodels from traces

For each scenario in the SDL model, an LQN submodel is extracted from its SDT trace in six steps:

- Step 1: convert the SDT trace into an angio trace.
- Step 2: identify the types (roles) of messages in the trace (Synchronous, Asynchronous, Reply, Forwarding).
- Step 3: identify the different services provided by each process.
- Step 4: find the precedence relationship between activities in each service.
- Step 5: handle the join events
- Step 6: map the software architecture model into an skeletal LQN submodel.

Figure 4.4 shows an example scenario trace, expressed for the present purpose by Message

Sequence Charts, which will be used to explain the above steps. The following sections walk through the model building process (the steps) as applied to the example.

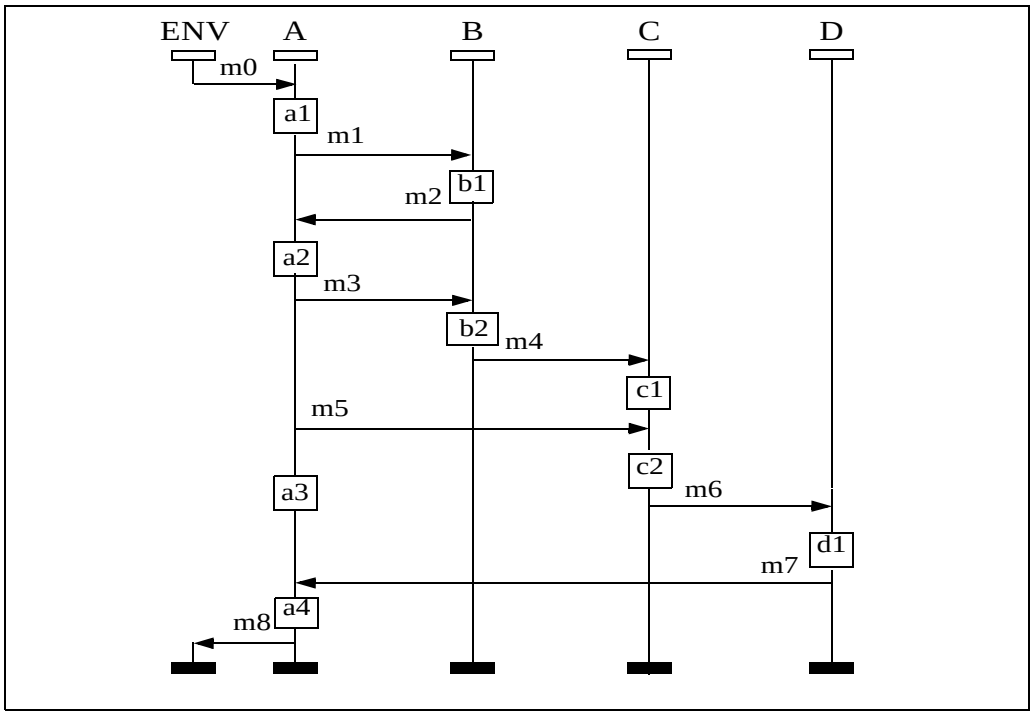


Figure 4.4 MSC of the example scenario to be modeled

Step 1: Convert an SDT trace into angio trace

The SDL tool we used, SDT [Telelogic96], can execute the specification and create an execution trace. For model building, the trace must record the sending and receiving of messages and the triggering of activities which consume resources such as CPU time. Each event must be associated with the process that executed it. Further, each message reception event must be traceable back to the event and the process that sent the message, so a chain of messages and execution can be captured in the model.

In SDT, trace messages are identified in such a way that the sending and receiving events could be matched up and activities are associated with the process that executed it. However, what is missing from the traces is the necessary information that specifies the end-to-end path of a response through the system and the causal relationship between activities in the system. This information is not recorded in the trace but it is possible to infer it. So, we post-process the traces in order to extract the required (missing) information and convert the traces into a form similar to angio traces, as described in [Hrischuk95].

An SDT trace is converted into a form similar to an angio trace in two steps. First the SDT trace is normalized. Then, the system actions are tagged with special identifiers, called “dye_ids”, that show their interconnection and relation to the trigger event.

Normalizing SDT traces

At the normalization step, the state-based SDT traces are converted to an activity_based intermediate trace form, as shown in Figure 4.5. The motivation for doing this is to record all attributes related to each activity in just one place (one record or one line). This will greatly simplify the model-building machinery.

For each activity the following attributes are identified. (1) activity name (e.g. send, receive, compute), (2) activity cost code that reflects the execution cost of the activity, (3) the active process that performed the activity, for example the sender process in case of a send activity, (4) the partner process, i.e the destination process in case of a send activity and the origin in case of a receive activity, (5) the name of the incoming message in case of a receive activity and the name of the outgoing message in case of a send activity.

```
*** TRANSITION START
*   PID    : A:1
*   State  : ready
*   Input  : m0
*   Sender : env:1
*   Now    : 0.0000
*   TASK  "a1"
*   OUTPUT of m1 to B:1
*** NEXTSTATE  ready

*** TRANSITION START
*   PID    : B:1
*   State  : ready
*   Input  : m1
*   Sender : A:1
*   Now    : 0.0000
*   TASK  "b1"
*   OUTPUT of m2 to A:1
*** NEXTSTATE  ready
```

(a) SDT state-based trace

active process	activity name	activity type	msg name	partner process	activity code
A1	RECEIVE	NEW_RCV	m0	env1	
A1	COMPUTE				a1
A1	SEND	ASY	m1	B1	
B1	RECEIVE	NEW_RCV	m1	A1	
B1	COMPUTE				b1
B1	SEND	ASY	m2	A1	

(b) activity-based trace

Figure 4.5 The normalization step

Assignment of "dye_ids" to messages

Dye_ids are defined and assigned to messages in such a way that allows processes to link the messages they send to the messages they have received. Dye_ids are defined using the following formula:

<dye_id> := <identifier> | / if the message is a triggering stimulus*/*
<input dye_id><separator><no. of msgs sent since input dye was received +1>
<separator> := .

The process of dye-id assignment starts with the assignment of some identifier (the initial dye_id) to the stimulus that initiated the trace. Then, to keep the causality across process boundaries, the process that receives the stimulus will use its dye_id as a base to construct the dye_id of the messages that it will send. Subsequent processes follow the same procedure for dye-ids assignment to causally link the message they send to the messages they have received. This allows us to infer dye_ids.

To implement the approach, we assign a counter to each process in the system to keep track of number of messages sent by the process. Each time a process receives a message, it resets its counter to zero, and stores the dye_id of the message received in a data structure called "current_dye". To construct the dye_id of an output message, the process first increments its counter, then concatenates the current_dye with the content of the counter and uses this construct as the new dye_id of the output message. The angio-trace corresponds to the SDT trace shown in Figure 4.5 (a) is given below in Figure 4.6. It is identical to the activity-based trace shown in Figure 4.5 (b) but with dye_ids.

active process	activity name	activity type	msg name	partner process	activity code	dye_id
A1	RECEIVE	NEW_RCV	m0	env1		1
A1	COMPUTE				a1	
A1	SEND	ASY	m1	B1		1.1
B1	RECEIVE	NEW_RCV	m1	A1		1.1
B1	COMPUTE				b1	
B1	SEND	ASY	m2	A1		1.1.1

Figure 4.6 The angio trace corresponds to the SDT trace in Figure 4.5 (a)

The result of applying the dye_id assignment technique to the trace shown in Figure 4.4 is depicted in Figure 4.7. In Figure 4.7, message m0 is assigned a dye_id of (1) because it is the first message sent from the environment (ENV). Upon receiving message m0, process A resets its counter to zero and assigns the dye_id of the incoming message, which is (1), to its current_dye. Then, to construct the dye_id of message m1, process A increments its internal counter and concatenates it with its current_dye to form the dye_id of m1 (1.1). When process B receives message m1, it resets its counter and use the dye_id of m1 as its current_dye. To, construct the dye_id of message m2, process B increments its counter and concatenates it to its current_dye and use this construct as the dye_id of m2 (1.1.1). The rest of the dye_ids are

generated in the same fashion. So, according to our technique for dye_id construction, a message is considered to be causally derived from another message appeared earlier in the trace if the dye_id of the earlier message is a prefix of the dye_id of the message.

The algorithm of dye_id assignment is listed in Section 4.5, algorithm "generate_message_id".

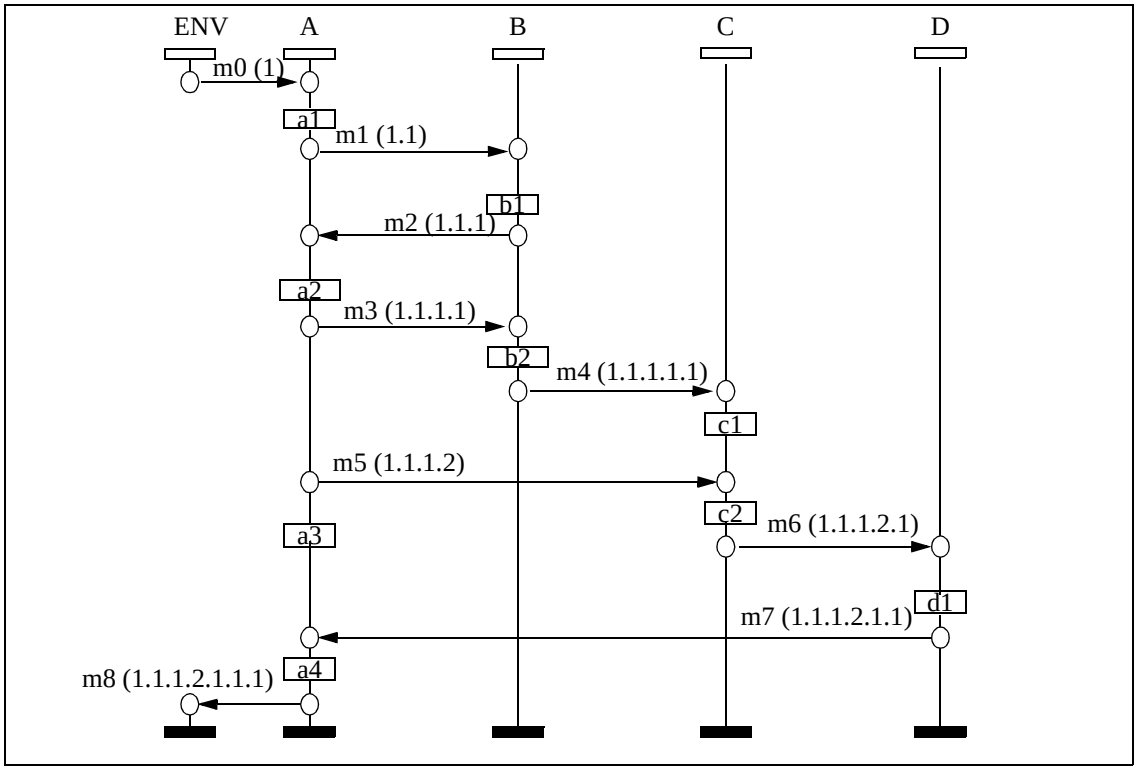


Figure 4.7 Dye-id assignments to messages in Figure 4.4

Step 2: Identify the message types (Sync., Async., Reply, Forwarding)

This step distinguishes between three types of messages sent by an SDL process; RPC-like, asynchronous and reply messages. A message is considered to be RPC-like if the process that sends the message, later on, receives another message that is causally derived from the message

that was sent. In this case, the received message will be identified as a Reply to the previous request. On the other hand, if the process does not receive any message that is causally derived from the message that was sent, the message sent is considered to be an Asynchronous message (ASY). RPC-like interactions are divided in step 4 into RPCs and Asynchronous RPCs.

Consider for example the scenario depicted in Figure 4.8. Message m1 is interpreted as an RPC-like message and m2 is identified as its Reply because process A, which sends message m1, later on, receives message m2 which is causally derived from m1. Causal linkage is detected by matching the dye-ids of the two messages (m1 and m2) and realizing that the dye-id of the message sent (m1) is a prefix of the dye-id of the message received (m2). On the other hand, message m3 is considered to be asynchronous because process A, which sent this message, doesn't receive any message that is causally derived from it.

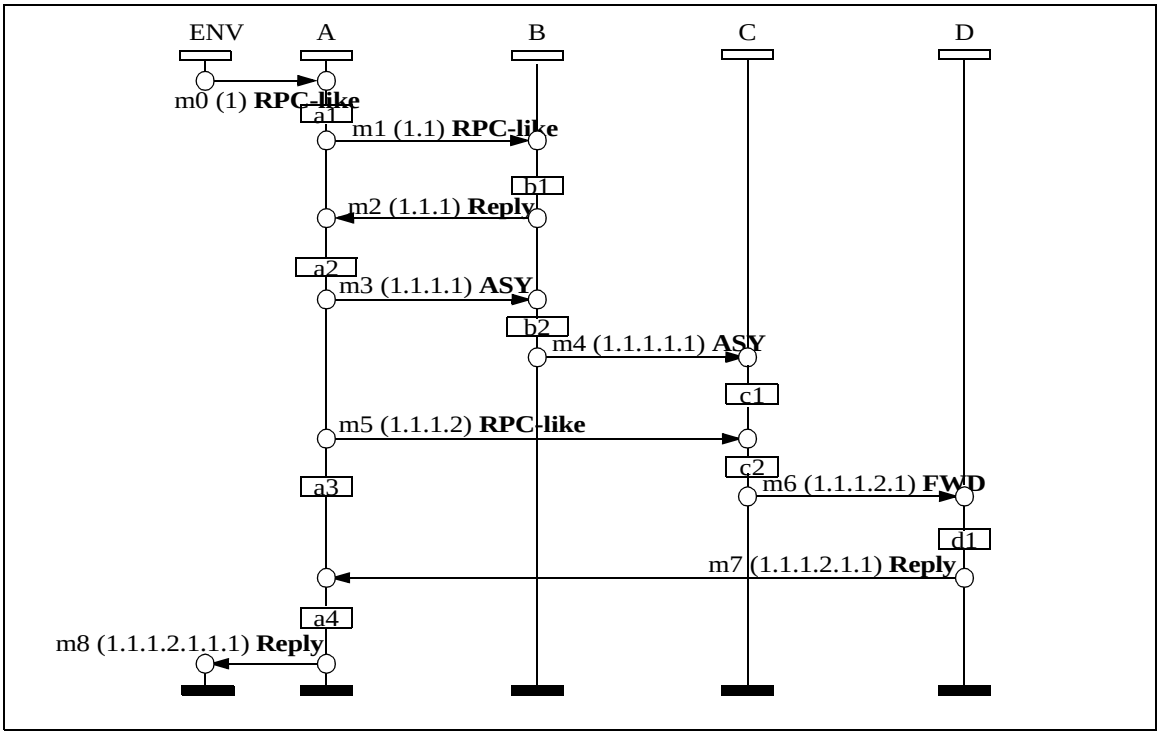


Figure 4.8 Message type identification

Some asynchronous messages play a forwarding role. To identify them we track each RPC request till the response returns to the sender. Then, we consider the asynchronous messages that belong to the path to be forwarding messages except the last message, that returns the response to the sender, which is the Reply message. In Figure 4.8, message m5 is an RPC-like request, m6 forwards it and m7 is the reply.

The notion of RPC used here does not block the process. Instead we assume that an RPC forks a thread which waits for the reply, leaving the process free (optionally) to do other things in the mean time.

The algorithm used to identify the type of messages is listed in Section 4.5, algorithm “identify_messge_types”.

Step 3: Identify the different services provided by each process

To identify the different services provided by each process and the group of activities associated with each service, we post-process the trace of each process separately. Each time a process receives a new request for service (RPC, Async. or Forwarding), we identify a new service for that process, and map all subsequent activities in the trace, till the reception of another service request, into this service. Note that the reception of a reply message does not begin a new service entry, it is a continuation of a previous service.

Once we partition processes into services, we update the attributes of the send activities in the traces so as to point to the proper target service. The algorithm for identifying and naming the different services of a process and updating the attributes of send activities is listed in Section 4.5, algorithm “identify_process_services”. Figure 4.9 shows the services associated with each process, with a dashed box corresponding to each service. It also shows the group of activities

within each service. The execution of a process is also shown as terminating at the end of each service.

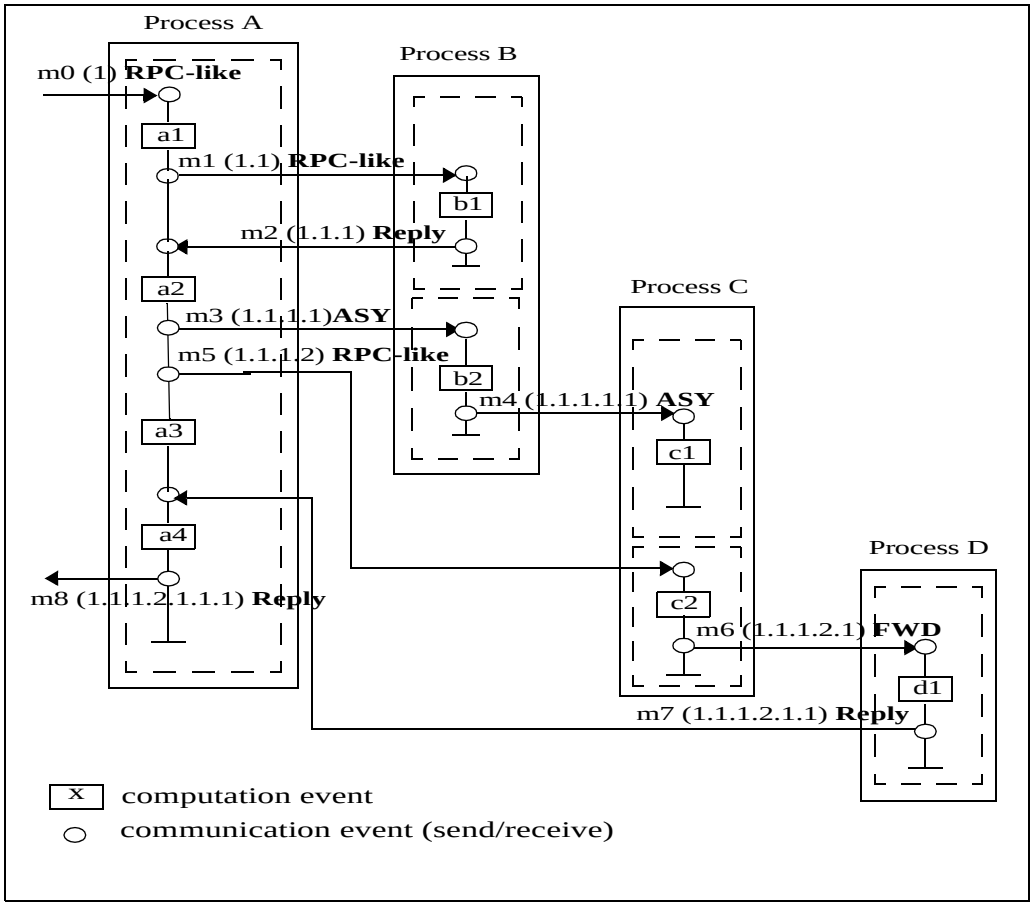


Figure 4.9 Services associated with each process, and activities within each service

Step 4: Find the precedence relationship between activities in each service

Within each service the precedence relationship among the activities is now recorded. Also, RPC-like interactions are separated into RPCs (in which the reply message immediately follows the request) and asynchronous RPCs (all other cases).

To describe an asynchronous RPC, we fork a subprocess to wait for the reply and let the main thread continue up to the event before the reply is received, where it terminates. The new sub-

process receives the reply and takes over the role of the main thread. For example, in Figure 4.9, message m5 is identified as an asynchronous RPC request and message m7 is identified as its reply. So, to model the semantic of this asynchronous RPC pattern, we fork a new subprocess at the location of the send activity. Activities that follow the send request (a3) remain connected to the current process, while the activity that receives the reply (m7) and the activities that follows it (a4 and the send activity of m8) are assigned to the new subprocess, as shown in Figure 4.10.

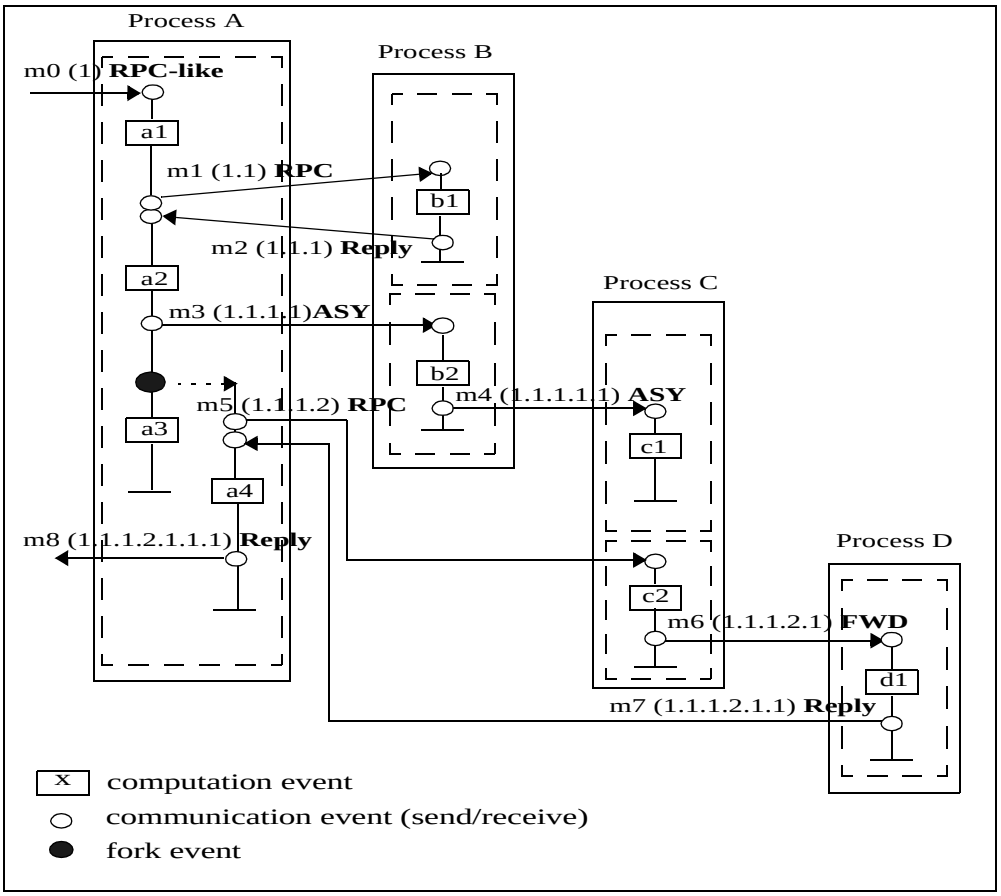


Figure 4.10 Software architecture with activity precedence information

Step 5: handle the join events

As we discussed in Chapter 3, the location of the join points and the threads to be joined are

identified using two special designer specified events “GOTO_JOIN” and “END_JOIN”. These events are recorded in the traces the same way as any other SDL standard event. The “GOTO_JOIN <join_name>” event is used to indicate that the current thread, which performed the event, will wait until all the requirements of the specified join point are satisfied. The “END_JOIN <join_name>” event is used to indicate that all requirements of the specified join point have been met. And as a result, all waiting threads will be merged into one thread which will continue the execution after the join point.

The mechanism for handling the join events in a process works as follows. First, we traverse the precedence graphs of the process, which we have constructed in the previous section, one at a time. Each time we encounter a “GOTO_JOIN <join_name>” event, we make a note of the location of that event and the name of the join point which it will be waiting at. Then when we detect an “END_JOIN <join_name>” event, we connect all activities that are waiting at this particular join point to it.

To keep the illustrating diagrams simple, there are no join points in our example scenario. However, some example traces that contain join events and their corresponding LQN models are shown in Chapter 3, Figure 3.10 and Figure 3.11.

Step 6: Map the software architecture into a skeletal LQN-submodel

Once we identify the different services associated with each process, and determine the precedence graph that describes their execution, mapping the software model into a LQN-submodel can be done straightforwardly by mapping each process in the software model into a process in the performance model and each group of activities in a service into a separate entry in that process. Each service precedence graph gives the activity precedence graph description of an entry. Within a service precedence graph, each activity is still an activity node in the

activity precedence graph. An activity node is inserted for each receive event, RPC pair, asynchronous send, and forwarding send. No nodes are inserted for reply events. Reply events are made to the entries in which they belong. Precedence relationships between the nodes follow from the flow of execution within the activities in the service precedence graphs. Figure 4.11 shows the LQN-submodel that models the software architecture depicted in Figure 4.10.

4.4 Merging the submodels and completing the model

If there is more than one LQN submodel, we merge these submodels into one LQN model by grouping the entries that have the same process identifier together as illustrated earlier in Figure 4.3. Then, to complete the performance model, the model user must populate it with additional information, which is not in the SDL specification, such as process allocation, workload parameters, resource demands for the activities identified in the traces and process priorities; some of these were illustrated in Figure 2.1. A textual interface has been designed to provide straightforward input of this information to the model, using the following sections.

i) Processor - allows user to describe the system's hardware components, and their scheduling discipline.
ii) Process - allows user to specify the allocation of processes to processors and define their priorities.
iii) Workload - allows the definition of the arrival rates of scenarios (stimulus).
iv) Performance Requirements - allows the user to specify the deadlines associated with critical scenarios (stimulus).
v) Resource functions (demands) - allows user to parametrize the activities available in the system by providing execution costs (in msec.) to them.

Resource demands of step (v) above can be found in several ways. They may be budget values assigned by a system planner, measured values from previous projects, or expert estimates of probable demand. Finding resource demands can require significant effort

[Menasce99]; however, this step is not considered here.

The formal description of the interface file in BNF form is shown in Figure 4.12.

This completes the construction of the performance model, which can now be solved to give performance evaluations. For example, the model in Figure 4.11 can provide predictions of the delay to receive a response to a request made by ENV to process A, of the probability of exceeding a deadline at this interface, of the mean waiting time of a message going to process B, or of the utilization of the processors on which the software runs.

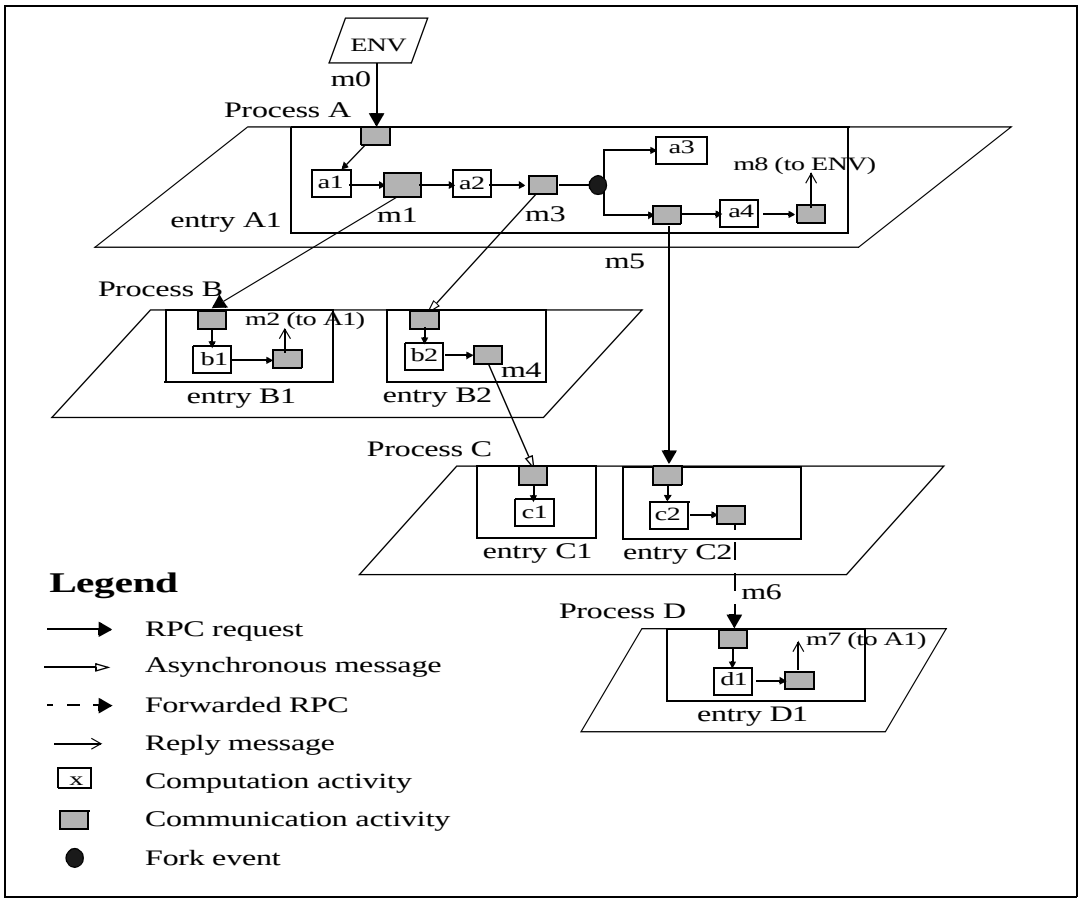


Figure 4.11 An LQN-submodel for the of the software architecture depicted in Figure 4.10

```
<interface_file>          == <processor_section> <task_section> <workload_section>
                             <perfRequirement_section> <resDemans_section>
<processor_section>       == <proc_sectionID> <proc_decl_list>
<proc_sectionID>          == <proc_section_keyword>  <end_line>
<proc_section_keyword>    == "Processor_Section"
<end_line>                == <CR><LF>
<proc_decl_list>          == <proc_decl> | <proc_decl> <proc_decl_list>
<proc_decl>               == <proc_id> <sheduling_flag> <end_line>
<proc_id>                 == <identifier>
<scheduling_flag>         == f          /* First come, first served */
                          |  p          /* Priority, premptive */
                          |  h          /* Head Of Line */
                          |  r          /* Random */
                          |  s          /* Processor sharing */
<task_section>            == <task_sectionID> <task_decl_list>
<task_sectionID>          == <task_section_keyword>  <end_line>
<task_section_keyword>    == "Task_Section"
<task_decl_list>          == <task_decl> | <task_decl> <task_decl_list>
<task_decl>               == <task_id> <task_priority> <proc_id><end_line>
<task_id>                 == <identifier>
<task_priority>           == <integer>      /* higher value for higher priority */
<workload_section>        == <workload_sectionID> <workload_decl_list>
<workload_sectionID>      == <workload_section_keyword>  <end_line>
<workload_section_keyword> == "Workload_Section"
<workload_decl_list>      == <load_decl> | <load_decl> <workload_decl_list>
<load_decl>               == <stimulus_id> <arrival_rate> <end_line>
<stimulus_id>             == <identifier>
<arrival_rate>            == <real>         /* arrivals per second */
<perfRequirement_section> == <perReq_sectionID> <perfReq_decl_list>
<perReq_sectionID>        == <perfReq_section_keyword>  <end_line>
<perfReq_section_keyword> == "PerformanceRequirement_Section"
<perfReq_decl_list>       == <perfReq_decl> | <perfReq_decl> <perfReq_decl_list>
<perfReq_decl>            == <stimulus_id> <max_response_time> <end_line>
<max_response_time>       == <real>
<resDemands_section>      == <resDem_sectionID> <resDem_decl_list>
<resDem_sectionID>        == <resDem_section_keyword>  <end_line>
<resDem_section_keyword>  == "ResourceDemands_Section"
<resDem_decl_list>        == <resDem_decl> | <resDem_decl> <resDem_decl_list>
<resDem_decl>             == <activity_id> <execution_cost> <end_line>
<activity_id>             == <identifierl>
<execution_cost>          == <real>
```

Figure 4.12 The BNF of the textual interface

4.5 Algorithms

The following algorithms implement the steps described in the previous sections.

Definitions:

traceFile = the file that stores the activities that took place during the execution of one scenario. Each activity is stored in one record.

actIndex = an index that identifies the location of activities in the trace file

actName(actIndex) = the name of the activity indexed by “actIndex”.

actType(actIndex) = the type of the activity indexed by “actIndex”.

- Activity RECEIVE can have two types (i) NEW_RCV, if the incoming message is a request for service (RPC, Asynchronous or Forwarding) (ii) REP_RCV, if the incoming message is a reply. Initially, the type of activity RECEIVE is set to NEW_MSG.
- Activity SEND can have four types: (1) ASY, if the message is an asynchronous message (2) RPC, if the message is an RPC message (3) FWD, if the message is a forwarding message, and (4) REP, if the message is a reply message. Initially, the type of activity SEND is set to UN_DEFINED.

activeProcess(actIndex) = the name of the process that performed the activity. For example, the sender process in case of a send activity and the receiver process in case of a receive activity.

partnerProcess(actIndex) = the destination process in case of a send activity and the origin process in case of a receive activity.

msgName(actIndex) = the name of the incoming message in case of a receive activity or the name of the outgoing message in case of a send activity.

msgDyeId(actIndex) = the dyeID of the incoming message in case of a receive activity or the name of the outgoing message in case of a send activity.

entryName(actIndex) = the name of the target entry in case of a send activity and the name of the new entry in case of a receive activity with type NEW_RCV.

Algorithm 1 Generate_message_id

```
Define: msgCount(p) = a counter for all messages sent out by process p.
        dyeRcvd(p) = the dyeId of the last message received by process p

Begin
    Initialize msgCount(p) = 0; for every process p in the system
    for actIndex =1 to LastActivity(traceFile)
       define currActivity = actName(actIndex)
       if currActivity is SEND
          sender = activeProcess(actIndex)       //define the sender process
          msgCount(sender) = msgCount(sender) + 1 // increment message counter.
          if (sender) is ENV
             then msgDyeId(actIndex) = msgCount(sender)
             else msgDyeId(actIndex) = dyeRcvd(sender)+'.'+msgCount(sender)
          endif
       endif
       if currActivity is RECEIVE
          receiver = activeProcess(actIndex)        // define the receiver process
          find in the trace-table the send activity, with an activity index
             "SendActIndex", that satisfies the following condition:
                ((msgName(SendActIndex) == msgName(actIndex)) &&
                (partnerProcess(SendActIndex) == activeProcess(actIndex)) &&
                (activeProcess(SendActIndex) == partnerProcess(actIndex)))
          if (no match was found)
             then report an error "inconsistent trace" and exit
             else  dyeRcvd(receiver)    = msgDyeId(SendActIndex)
                   msgDyeId(actIndex) = msgDyeId(SendActIndex)
          endif
       endif
    endfor
End
```

Algorithm 2 Identify_message_types

```
Begin
    Initialize DyesSent to {}
    for actIndex =1 to LastActivity(trace_file)
        Let  currActivity = actName(actIndex)
        if currActivity is SEND then
            Let currDyeId  = msgDyeId(actIndex)
            actType(actIndex) = UN_DEFINED
            append currDyeId to DyesSent
            senderIndex(currDyeId) = actIndex    //store the index of the current activity
        endif
        if currActivity is RECEIVE then
            currDyeId  = msgDyeId(actIndex)
            actType(actIndex) = NEW_RCV        // the default type of activity receive is NEW_RCV
            let receiver = activeProcess(actIndex)
            search the set of DyesSent for a "sourceDyeId" that satisfies the following condition:
                ((sourceDyeId is is a prefix of currDyeId) &&
                 (receiver == activeProcess(senderIndex(sourceDyeId)) &&
                 (actType(senderIndex(sourceDyeId)) has not been defined yet)).
            If (match was found)
            then  // the message received is a REPLY.  So. an RPC pattern is detected.
                  //Update the type of the receive activity to REP_RCV, and  send activity to RPC//
                  actType(actIndex) = REP_RCV
                  actType(senderIndex(sourceDyeId)) = RPC
                  // Update the type of the intermediate send activities along the RPC path
                  // to type FWD except the type of the last send activity, update its type to REP.
                  Repeat for every "dyeId" in the set of DyesSent
                     if  ((sourceDye is a prefix of dyeId) && (dyeId is a prefix of currDyeId) &&
                        (actType(senderIndex(dyeId)) has not been defined yet))
                     then  // dyeId belongs to the the RPC pattern.
                        if (dyeId = currDyeId)
                           then   //  last message in the RPC pattern.
                                  //  Change the type of the send activity to REP.
                                  actType(senderIndex(dyeId)) = REP
                           else   // an intermediate msg. in the RPC pattern.
                                  // Change the type to the send activity to FWD
                                  actType(senderIndex(dyeId)) = FWD
                           endif
                        endif
                  endRepeat
               endif
            endif
        endfor
    set the type of all UN_DEFINED send activities to ASY
End
```

Algorithm 3 Identify_process_services(entries)

```
Begin
    Initialize serviceCount(p) = 0; for every process p in the system
    for actIndex = 1 to LastActivity(trace_file)
        if ((actName(actIndex) is RECEIVE) && (actType(actIndex) is NEW_RCV))
        then    // assign a new service to the current process.
            receiver = activeProcess(actIndex)          //define the receiver process
            currDyeId = msgDyeId(actIndex)
            ++serviceCount(receiver)
            serviceName(actIndex) = receiver + "_"+serviceCount(receiver)
            targetService(currDyeId) = serviceName(actIndex)
        endif
    endfor
    for actIndex = 1 to LastActivity(trace_file)
        if (actName(actIndex) is SEND)
        then  serviceName(actIndex) = targetService(msgDyeId(actIndex))
        endif
    endfor
End
```

Chapter 5: The Optimization Strategy

5.1 Introduction

This chapter describes an optimization technique for obtaining compliance with response deadlines. A design is considered to be *compliant* to all requirements (and thus, feasible) if, for every scenario, the probability that its response time exceeds its deadline is less than some threshold; this threshold can be set to zero in hard real-time systems.

The technique improves a non-compliant design by making incremental changes to the design. The approach taken is to identify the factors that might contribute to the problem, rank them, and from the ranking generate a design change. The candidate factors to be considered are the task priorities, the task allocations to processors, and the constraints imposed by the task structures (which will be altered only to a limited degree).

The proposed methodology consists of two phases: An initialization phase and an optimization (refinement) phase. The goal of the initialization phase is to efficiently find a good initial configuration or solution (task allocation and priority assignment) for the design. The goal of the optimization phase is to improve the initial design in a series of steps, and make it compliant with all requirements.

To improve the performance of a non-compliant design, the contributor(s) for performance problems must first be identified so that proper design change(s) can be recommended. The

possible contributors considered here are wrong design decisions on task priority, task allocation and/or task structuring,

Identifying the most significant contributor, and consequently the recommended design change, is difficult because of the close interaction and trade-offs between these design decisions. For example, reallocating a task T to a processor P will increase the computational load on P, but may reduce the communication overheads (if T communicates with tasks on P), hence reducing the response time. The trade-offs can become very complex as the real-time architecture becomes more expressive; so a simple and scalable approach is required.

In this research, the design optimization process is formulated as a global optimization problem and an incremental heuristic technique is proposed to solve it. At each step, the design is evaluated. If the design is non-compliant with any of the requirements, possible contributing factors are analyzed, a key factor is determined and some priority changes are recommended. The new configuration is then evaluated and the quality of the solution is estimated. This is repeated, as shown in Figure 5.1, until either a feasible solution is found or some stopping criteria are satisfied. If the search on priorities converges without obtaining feasibility, then a structural change is introduced (a design reshaping), and the search is restarted for priorities.

```
Begin    // Procedure for building feasible real-time systems.
    Phase1: Initialization
      - Find an initial task allocation and priority assignment
      - Evaluate the initial design (solve LQN model)
      - If all scenarios are compliant with performance requirements then STOP
      - Estimate solution quality.
    Phase2: Optimization
      - Repeat  // Refine design configuration.
        - Identify key factor.
        - Recommend a configuration change.
        - If (no configuration change can be generated) reshape design
        - Evaluate recommendation (solve new LQN model)
        - If  recommendation leads to a compliant system EXIT.
        - Estimate solution quality.
        - If  (solution quality improves)
              - save current (best) design configuration.
          else
              - if (the search converged without obtaining feasibilities)
                  - restore best design configuration.
                - reshape design.
              endif
          endif
      - Until (some predefined stopping criterion has been met)
End // Procedure
```

Figure 5.1 A procedure for building feasible real-time systems

5.2 Finding an Initial Design Configuration

The goal of this phase is to determine a good task allocation and priority assignment for the design. The task allocation subproblem is solved first, then the priority assignment subproblem. However, instead of attempting to find the “best solution” for each subproblem, we focus on efficiently determining a reasonably good initial solution to the combined problem.

We use the MULTIFIT-COM allocator [Woodside93] to solve the task allocation subproblem, and a simple deadline-based priority assignment method to assign priorities to tasks

[Sun96]. In the deadline-based methods, a priority is assigned to a task inversely proportional to its relative deadline as was discussed in Section 2.4.3.2.

5.2.1 Finding an initial task allocation using MULTIFIT-COM

The MULTIFIT-COM algorithm allocates a single application system in such a way to maximize system throughput (minimize the computational and communication resource requirements for tasks in the system), by reducing bottlenecks. Although the allocator does not consider schedulability issues during allocation, we use it based on the observation that faster systems (i.e., ones which maximize system throughput), or systems that have balanced load, are more likely schedulable [Chu89], [Santos97], [Houstis90].

As MULTIFIT-COM is designed to allocate a single application system, it assumes that all tasks in the system have the same invocation rate and each task is visited only once during the application cycle. Hence, it uses the computation time of tasks as estimate for task sizes, and the total communication cost between two tasks as an estimate for the communication overheads. In the case of LQN models, however, a model can include several applications (scenarios), with different invocation rates, that can run concurrently. So it is very possible to find tasks in LQN models that are involved in many scenarios (each scenario may have a different invocation rate). Moreover, the number of visits to tasks are not the same in the model, some tasks may be visited more than others. Accordingly, to be able to allocate LQN models using MULTIFIT-COM, we need to use a different measure for task sizes and communication overheads between tasks

This work uses the total workload on a task as an estimate for its size and the total overheads due to message exchange between any two tasks as an estimate to the communication overhead between them. The metric for estimating the size of task T is given below in equation (5.1) and the metric for estimating the communication overheads between tasks T1, T2 is given in

equation (5.2).

$$taskSize(T) = \sum_{\forall e \in entries(T)} executionDemand(e) \times invocRate(e) \qquad (5.1)$$

$$executionDemand(e) = \sum_{\forall activity \in \text{ activities}(e)} executionTime(activity)$$

$$invocRate(e) = \begin{cases} arrivalRate(e) & \text{if e has an open arrival} \\ 1/peroiod(e) & \text{if e is periodical} \\ \sum_{\forall e_0 \in \text{ predecessor(e)}} NoMsgs(e_0,e) \times invocRate(e_0) & \end{cases}$$

where predecessor(e) is the set of entries that calls e

$$CommOV(T_1, T_2) = \sum_{\forall e \in entries(T_1)} CommOV(e, T_2) + \sum_{\forall e \in entries(T_2)} CommOV(e, T_1) \qquad (5.2)$$

$$where \quad CommOV(e, T) = \sum_{\forall m \in \text{ messages from e to T}} commCost(m) \times invocRate(e)$$

In equations (5.1) and (5.2), the set of entries of task T is $entries(T)$, and an entry e is regarded as a set of activities $activities(e)$. The invocation rate of an external entry $invocRate(e)$, which is triggered by an external event, is set equal to the arrival rate or 1/period of the event (message) that triggers the entry. The invocation rate of an internal entry, which is triggered by a message sent by other entry(ies) in the model, is determined by a special recursive equation as shown in equation (5.1). The term $executionTime(activity)$ represents the

execution cost (demand) of the activity in msec. The term $commCost(m)$ represents the overhead (cost) in msec. associated with sending and/or receiving message m.

Once the task size and CommOV values are obtained, MULTIFIT-COM can be applied with one of the eight policies discussed in Section 2.4.2.1. The MULTIFIT-COM allocator was used here because it is efficient as well as available.

5.2.2 Finding an initial priority assignment

Initial priorities are assigned to tasks using a simple deadline-based method, called "Proportional-Deadline" [Sun96], which assigns priorities to tasks according to their proportional deadline. The longer the proportional deadline of a task, the lower the priority of the task. The proportional deadline of a task is obtained by dividing the global deadline of its parent transaction among its tasks proportionally to their execution times.

The "Proportional-Deadline" method was used here because it is efficient as well as simple to implement.

5.3 Evaluating a System Design

In this step the performance model of the design is solved by simulation. In this work the models are all layered resource models, and the simulation component of the Layered Queueing Network Solver (LQNS) [Franks95] was used to get the performance metrics. Then the probability that the estimated response time of each scenario (thread) exceeds the corresponding end-to-end requirement value is checked; the end-to-end performance requirements (deadlines) of scenarios are specified by the system users using the textual interface of the model-builder tool, as was described in Chapter 4. The probability of exceeding deadline is estimated for each

scenario to a specified percentage accuracy (given as a 95% confidence interval).

A system is asserted to be non-compliant if at least one scenario in the system is non-compliant. A scenario is considered to be non-compliant with its performance requirements if the probability that the computed scenario response time exceeds the scenario response time requirement is greater than the pre-specified threshold value. The threshold value would be set to zero in hard real-time systems.

5.4 Estimating the Solution Quality

In many systems, a design can be evaluated by the magnitude of a single number, for example the total execution time of all scenarios. Here, however, performance can not always be characterized by a single number, since there can be several responses, each with its own deadline to meet and the satisfaction of these deadlines is more important than shorter overall average delays.

In this thesis, we propose a penalty measure (criterion) which estimates how far the entire design is from being feasible. The measure is used at each step by the search algorithm to determine whether it is moving in the direction of performance improvement or not. The penalty is zero for a feasible design.

The penalty measure is calculated by summing up the criticality metric of all scenarios in the system as shown in equation (5.3), in which the criticality of a scenario defines the penalty assigned to the scenario when it does not comply with its deadline requirement. The behavior of the criticality metric is shown in Figure 5.2. The exponential power is used to emphasize the penalty assigned to scenarios with bigger probabilities of exceeding deadline. The bigger the probability that a scenario response time exceeds its deadline, the higher the value (penalty) that

will be assigned to its criticality metric. Furthermore, the parameter β is used to further emphasize the penalty assigned to scenarios with bigger probabilities of exceeding deadline. On the other hand, if the probability that a scenario response time exceeds its deadline is found to be zero, the scenario will be identified as an uncritical scenario and its criticality metric will be set to zero.

$$SolutionPenalty = \sum_{\forall(S \in Scenarios)} Criticality_S \tag{5.3}$$

$$Criticality_S = \begin{cases} 0 & probExceedDeadline \le \alpha \\ e^{\beta \times ProbExceedDeadline} & probExceedDeadline > \alpha \end{cases}$$

where α, β are design decision parameters. For example $\alpha = 0.05$ $\beta = 15$

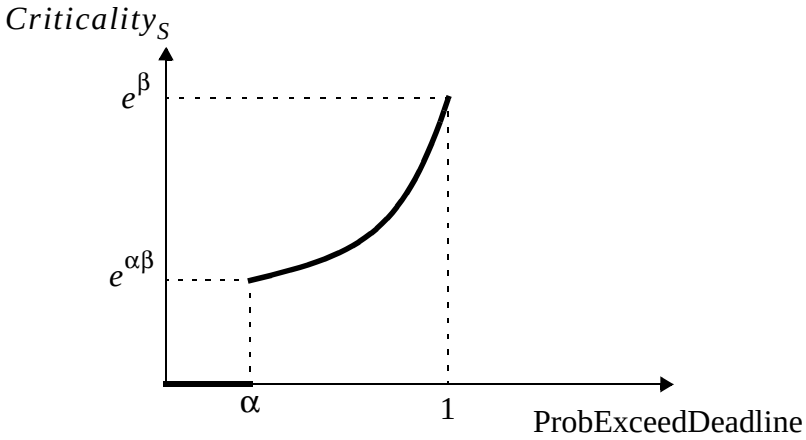


Figure 5.2 The behavior of the Criticality metric

The penalty measure tells how far the design (solution) is from being feasible. The smaller the penalty measure the closer the solution is to the optimal solution. A penalty measure of zero indicates a feasible (compliant) design.

5.5 Design Optimization

As we discussed earlier an unfortunate design decision on assignment of task priority, task allocation and/or task structuring could be the possible contributor for performance problems. In this work, first we will concentrate on diagnosing and improving performance problems by adjusting priority assignment until the solution converges without obtaining feasibility. Then use the other two design decisions (i.e. task allocation and structuring) to reshape the design, and the search is restarted again for priorities. The motivation for doing this is that improving on an existing priority assignment will not affect the decisions made earlier on task allocation and/or task structuring. This non-interacting property will allow us to easily isolate the bottlenecks due to priority assignment and make our adjustment accordingly.

The following subsections discusses the various steps in the optimization strategy, namely the priority adjustment process and the reshaping strategy.

5.5.1 Design improvement using priority adjustment

To adjust priorities to improve the value of SolutionPenalty, this work identifies a task whose priority can usefully be raised. A trivial strategy, to consider every task, would be wasteful. A random choice would make slow progress. The strategy we propose here for identifying the tasks, whose priorities need to be changed, is to concentrate on the tasks that were involved in performing the non-compliant scenarios and consider them as contributors for problem failures. To further identify the most significant task, a metric ($TaskMetric_T$) is defined in Equation (5.4) below to estimate the significance of task T. The metric combines values of $TaskScenarioMetric|_T^S$, which measure the delays and overheads in T during the execution of scenario S, including a weighting factor which emphasizes scenarios with deadlines misses.

The metric is also normalized by task utilization U_T to emphasize the impact on tasks with low utilization.

$$TaskMetric_T = \frac{1}{U_T} \times \sum_{\forall S \in Scenarios} TaskScenarioMetric|_T^S \times Criticality_S \qquad (5.4)$$

$$TaskScenarioMetric|_T^S = TaskWaitMetric|_T^S + TaskCommOVExcessMetric|_T^S$$

$$TaskWaitMetric|_T^S = \sum_{\forall (a \in Activities|_T^S)} WaitingTime_a$$

$$TaskCommOVExcessMetric|_T^S = \frac{\sum_{\forall (m \in nonLocalMsgs|_T^S)} CommOV_m}{noTargetCPUs|_T^S} - \sum_{\forall (m \in LocalMsgs|_T^S)} CommOV_m$$

Where

* $Activities|_T^S \equiv$ *set* of *activities* of task T that are on the critical path of scenario S
* $nonLocalMsgs|_T^S \equiv$ set of *non–local messages* of task T that are on the critical path of scenario S
* $LocalMsgs|_T^S \equiv$ set of *local messages* of task T that are on the critical path of scenario S
* $noTargetCPUs|_T^S \equiv$ no of CPUs that task T communicates with during the execution of the critical path of S

The measured values, per scenario in task T, are the total waiting time at activities in T, along the critical path of the scenario ($TaskWaitMetric|_T^S$) and the communication overhead excess due to inter-processor communication ($TaskCommOVExcessMetric|_T^S$).

The ($TaskCommOVExcessMetric|_T^S$) is the difference between two components. The

first component is the average cost of communication by task T with all CPUs other than the one which is already assigned to, and represents the maximum amount that the communication costs of T can decrease if the task is reassigned to another CPU. The second component represents the cost of communications by T with all tasks currently assigned to the same CPU as T, and represents additional communication costs that T will incur if moved to another CPU.

The taskMetric characterizes the impact of the weight of task T (i.e. waiting delays and communication overheads) on the response time of scenarios. The higher the value of the metric, the worse (the more significant) is the effect of task T on scenarios response times. Accordingly the critical performance factors are the tasks that ranked high according to the task metric, and the recommendation that gives the highest reward towards performance improvement is to increase the priority of the most significant (critical) task. This recommendation was chosen after performing some exploratory work with more tasks from the ranking without observing obvious benefit.

There is no proof that this technique always work; however, extensive experience gives some confidence as will be seen in Chapter 6.

5.5.2 Design reshaping

Once you have done your best with priorities, if more improvement is needed, other factors must be changed to reshape the design. In this work, the design was “reshaped” by reallocating and/or splitting up some tasks and the priority search was restarted. Design reshaping was needed in two situations: (1) when the candidate (most significant) task identified for priority adjustment was already the highest priority task on the CPU, or (2) when the solution quality did not improve after N priority adjustment steps, where N is a design decision parameter. N was assigned the value 6 in this research.

The strategy we propose here to reshape the design is shown in Figure 5.5. First we try to reallocate the most significant task to another CPU. If the reallocation process is not feasible because of some allocation constraints or if the task has been reallocated before then we try to split the task. If task splitting strategy is not feasible, because for example the candidate task has only one entry, then we consider the next candidate task. The reshaping phase iterates between these steps, as shown in Figure 5.5, until a reshaping step is made or until there is no more candidate task to consider.

5.5.2.1 Reshaping using task allocation

Task allocation is concerned with the assignment of tasks to processors. Alternative assignment schemes may affect many different performance attributes, including thread service times, processor utilizations, and intertask communication overheads.

The task that will be selected for reallocation (the candidate task) is the one that ranks first according to the TaskMetric criterion (equation 5.4), does not have any allocation constraint, and has not been reallocated before. This task will be allocated to the CPU which ranks last according to CPU metric, which is defined in equation (5.5).

$$CPUMetric_C = \sum_{\forall(\text{T allocated on C})} TaskMetric_T \times Util_C \qquad (5.5)$$

The CPU metric for CPU “C’ is based on the summation of the TaskMetric (equation 5.4) of all tasks on “C”, weighted by the current utilization of “C’, $Util_C$. Accordingly the higher the value of the CPU metric, the higher the risk of missing deadlines if the candidate task is allocated on the CPU. Thus, the candidate task is always allocated to the CPU which rank last according

to the CPU metric, provided that it is not prevented by an allocation constraint.

5.5.2.2 Reshaping using task restructuring

Task structuring is concerned with packaging software functions into tasks. Task structuring has a great impact on performance. It affects the degree of parallelism that can be achieved and also has an effect on priority assignment and task allocation. More importantly, a wrong design decision on task structuring, namely packaging critical and non-critical functions into one shared task, can introduce a phenomenon that is similar to the priority inversion. In priority inversion, a higher priority task is forced by some mechanism to wait for a lower priority task to complete.

Priority inversion can happen where multiple scenarios, with different real-time requirements, share a common task as shown in Figure 5.3(a). Suppose the path leading from event e1 is critical and all the tasks along that path are high priority tasks. However, the path can still suffer from an excessive queueing delay at task T5, if task T5 is busy serving less critical requests submitted earlier by other tasks in the system. Priority inversion can also happen within the execution of a single scenario, when multiple threads with various real-time requirements share a common task. Consider for example the design shown in Figure 5.3(b). Because the request to entry c1 in task C is sent before the request to entry c2, entry c1 will always be called before entry c2. Accordingly, entry c2 will always suffer from waiting times at task C regardless of its priority level. So, a good design decision to be considered in such cases is to restructure (split) the shared task C into two or more sub-tasks and give a high priority to the more critical subtask(s).

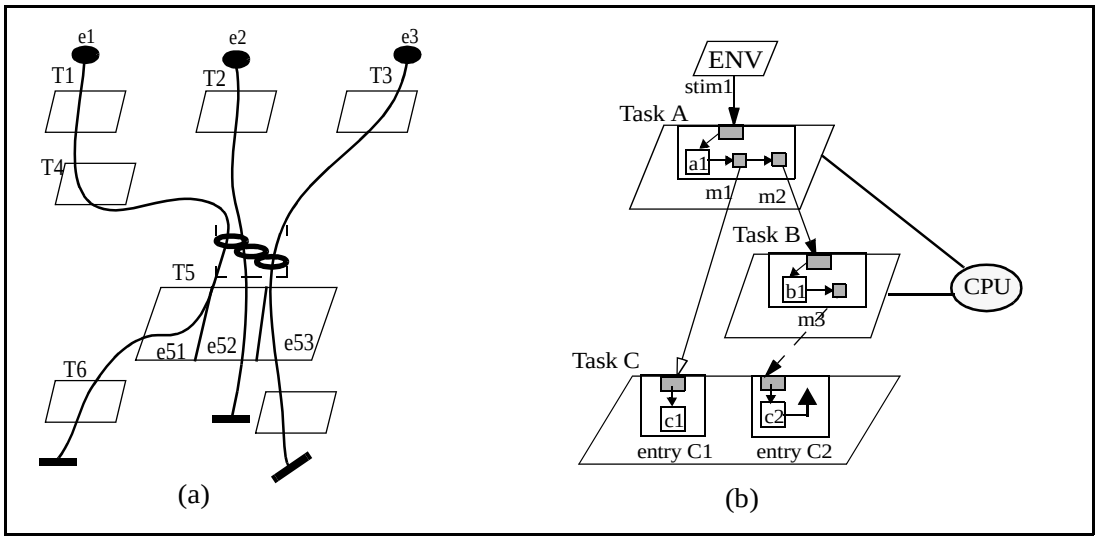


Figure 5.3 An illustration of the priority inversion problem during the execution of multiple scenarios (a), or a single scenario (b)

The strategy proposed for task structuring tries to eliminate the situations that may cause the priority inversion problem to appear. It splits a candidate task into two subtasks such that the first subtask contains the entry that suffers the most from waiting time (i.e. ranked first according to entry metric, equation (5.6)) while the second subtask contains the remaining entries.

$$entryMetric\big|_e^S = \sum_{\forall(a \in Activities\big|_e^S)} WaitingTime_a \times Criticality_S \qquad (5.6)$$

The priorities of the generated subtasks are derived from the priority of the original (parent) task as follows. If the priority level of the parent task is p, assign a priority level of p+1 to the first subtask, that includes the critical entry that experiences excessive waiting, and a priority level p to the second subtask.

By splitting a suspect task and adjusting the priority of the generated subtasks as was

described above, waiting times at critical entries will be eliminated if they were due to queueing at the original task. Also, even if this was not the case, the finer granularity of the generated subtasks will enhance the degree of parallelism that can be achieved. In addition, it will give more flexibility to the search technique in finding a good task allocation and priority assignment.

5.5.3 Summary of the design optimization process

Figure 5.4 shows the 4 main stages of the optimization strategy as well as the interaction between them. A more detailed flowchart of the optimization process is shown in Figure 5.5.

The purpose of each stage is given below.:

- The “Initialization” stage. This is an optional stage and used to find a good initial task allocations and priority assignmnets if they were not specified in advance.
- The “Evaluate Model” stage. This stage solves the LQN model and checks its feasibility. If it finds the solution feasible, it terminates the optimization process with success. Otherwise, it estimates the solution quality. If it finds that the solution quality has improved over that of the last step, it transfers control to the “Recommend Config. Change” stage. Otherwise, depending on some criteria, control may be transferred to the “Reshape Design” or “Recommend Config. Change” stage.
- The “Recommend Config. Change” stage. This stage sorts tasks according to taskMetric (equation 5.4) and increases the priority level of the most significant “candidate” task.
- The “Reshape Design” stage. This stage first tries to reallocate or split the candidate task. If it succeeds, it switches control to the “Evaluate Model” stage. Otherwise, it will consider the next task in the sorted list. This process is repeated until one task is reallocated or splitted. If the sorted list is exhausted without success, the optimization strategy will be terminated with Failure.

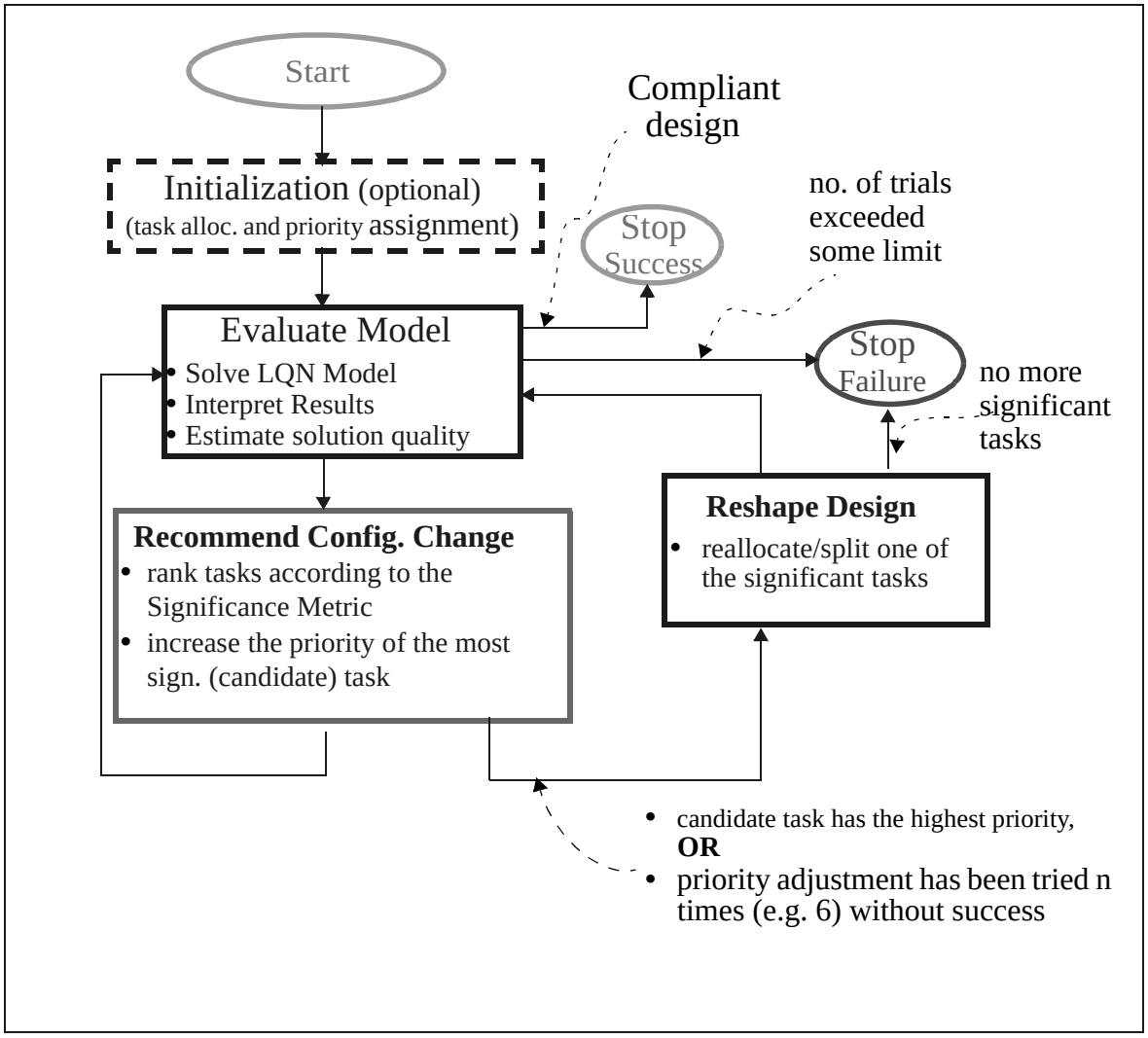


Figure 5.4 The 4 main stages of the optimization strategy

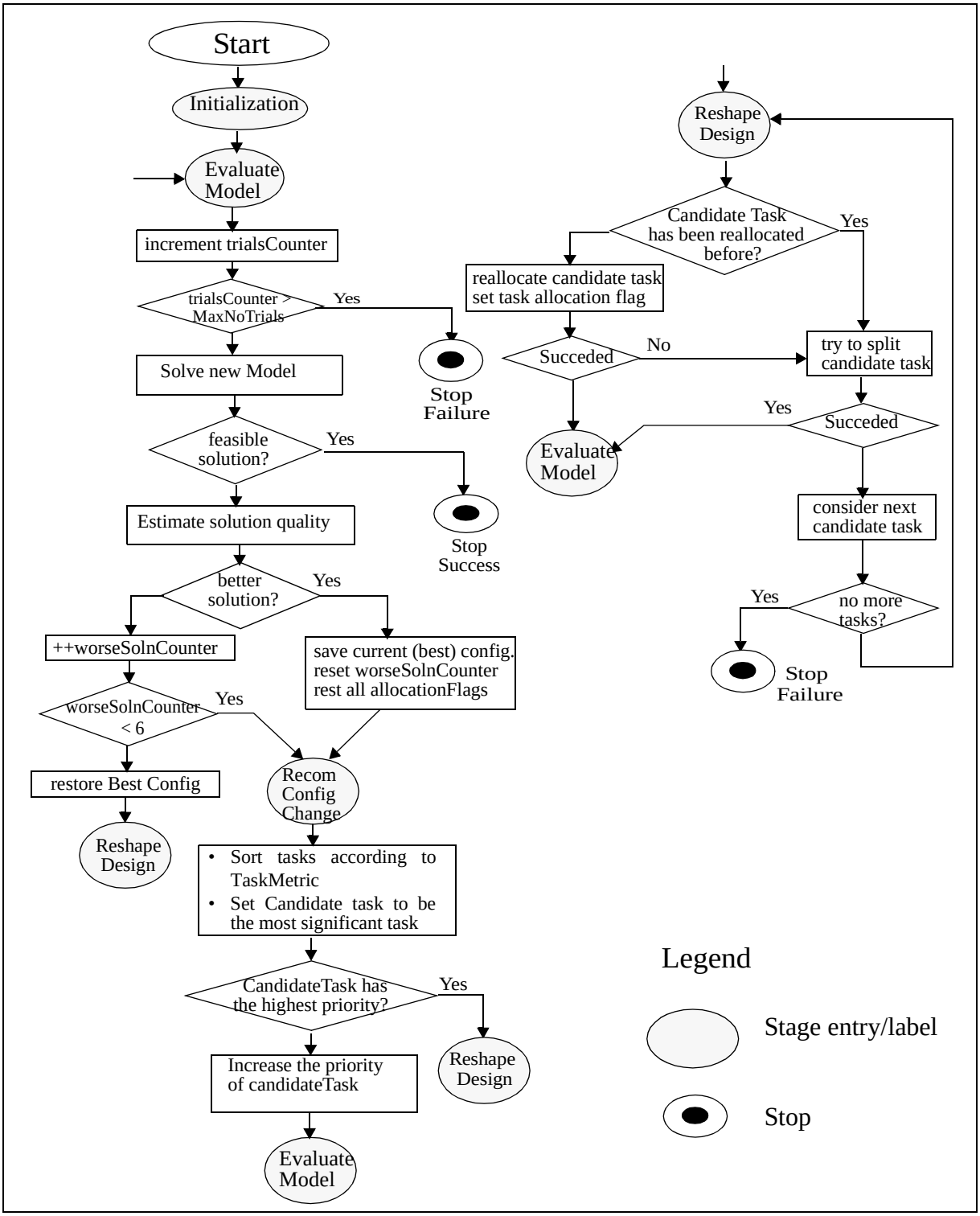


Figure 5.5 A detailed flow chart for the design optimization

5.6 A Tutorial Example

In this section we will demonstrate the application of the optimization steps in Figure 5.5 on the application example shown in Figure 5.6. In this example, the initial task allocation and priority assignment were done manually in such a way to illustrate some features in the improvement steps. For example, the priorities were assigned to processes in the reverse order of criticality which represents the worst case scenario as far as the priority assignment is concerned.

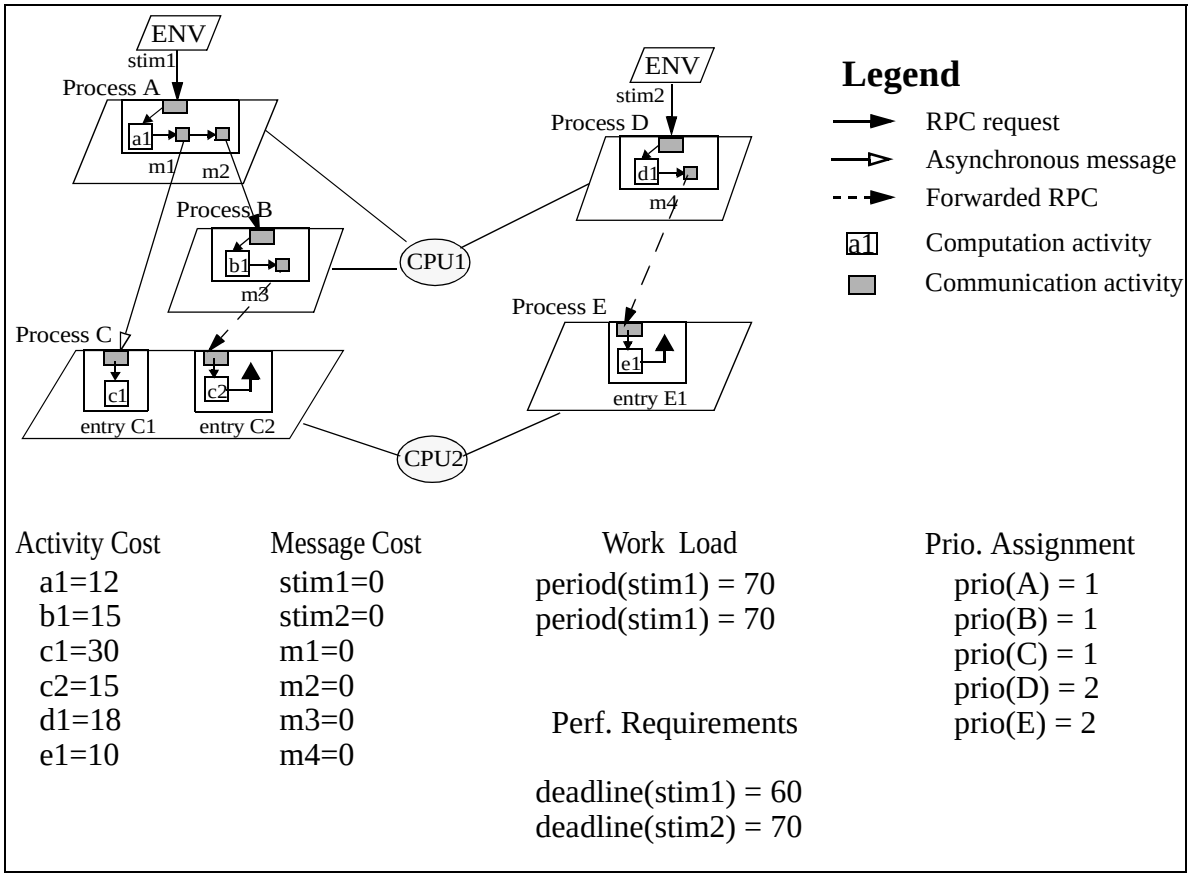


Figure 5.6 A tutorial example

5.6.1 Experiment #1

In this experiment, the model (shown in Figure 5.6) was solved using the LQN solver. The results are given as step 0 in Table 5.1, and show a response time to stim1 of 70 msec, which is non-compliant because it exceeds its requirements of 60 msec. The response time to stim2 is compliant. The optimization strategy was applied to the non-compliant design and managed to find a feasible design in 6 steps as shown in Steps 1 to 6 in Table 5.1, using only changes in priority. The algorithm did not apply re-allocation or task restructuring because the number of steps that gave no improvement did not exceed the threshold (set at 10 steps).

step	Candidate task	Action	Resp. times to stim1 & stim2	Prob. of missing deadline	Solution penalty metric Eqn (5.3)
0			*70.004* *28.000*	*1.000* *0.000*	*3269017*
1	*A*	*Adjust Priority* *current Priority order: D > A = B* *new priority order : D > A > B*	*70.004* *28.000*	*1.000* *0.000*	*3269017*
2	*A*	*Adjust Priority* *current Priority order: D > A > B* *new priority order : D = A > B*	*70.004* *28.011*	*1.000* *0.000*	*3269017*
3	*A*	*Adjust Priority* *current Priority order: D = A > B* *new priority order : A > D > B*	*67.000* *40.000*	*1.000* *0.000*	*3269017*
4	*B*	*Adjust Priority* *current priority order : A > D > B* *new priority order : A > D = B*	*67.000* *40.000*	*1.000* *0.000*	*3269017*
5	*B*	*Adjust Priority* *current priority order : A > D = B* *new priority order : A > B > D*	*67.000* *55.000*	*1.000* *0.000*	*3269017*
6	*C*	*Adjust Priority* *current priority order : E > C* *new priority order : E = C*	*57.000* *67.000*	*0.000* *0.000*	*0*

Table 5.1 The optimization steps of Experiment #1

5.6.2 Experiment #2

In this experiment the problem of achieving feasibility is made more difficult than in Experiment #1, by increasing the execution demands along the critical path of one scenario (or in other words to reduce the laxity of the scenario) on the behavior of the optimization method.

The cost of activity c2, in the model shown in Figure 5.6, is increased to 20 to reduce the laxity of the stim1 scenario. When the model was solved, the results were found not compliant with the performance requirements, as shown in Step 0 in Table 5.2, because the response time of stim1 exceeds its deadline requirement. The optimization strategy was applied to the non-compliant design and managed to find a feasible design in 10 steps as shown in Steps 1 to 10, in Table 5.2. During the first 7 steps, the optimization procedure was mainly applying the priority adjustment strategy. However, at step 8, a reshaping step was required because the candidate task selected for priority adjustment (task C) is already the highest priority task on its CPU (CPU2). The reshaping mechanism tried first to reallocate the task to CPU1, however it did not implement this option because it will increase the utilization on CPU1 beyond the threshold limit (set at 95%). So, the reshaping mechanism chose to split task C instead into two tasks C_1, C_2, and assign task C_1, which contains the most critical entry “entry C2”, a higher priority than C_2. After the reshaping step, the optimization procedure iteratively applied the priority adjustment strategy until it found a feasible solution as shown in Steps 9 and 10 in Table 5.2.

5.6.3 Experiment #3

This experiment has communication overheads when sending and receiving messages. An overhead cost of 0.1ms is included for each message, at both the sending and the receiving process. Table 3 shows the results. Again, the model was initially found not compliant with the performance requirements. The optimization procedure applied the priority adjustment strategy

during the first 4 steps. At Step 5, a reshaping step was required because the candidate task selected for priority adjustment (task E) was already the highest priority task. The design was then adjusted, by reallocating task E, and the optimization procedure iteratively applied the priority adjustment procedure until a feasible design was found.

Step	Cand. task	Action	Resp. times to stim1 & stim2	Prob. of missing deadline	Solution penalty metric Eqn (5.3)
0			*70.00* *28.00*	*1.000* *0.000*	*3269017*
1	*A*	*Adjust Priority* *current Priority order: D > A = B* *new priority order : D > A > B*	*70.00* *28.00*	*1.000* *0.000*	*3269017*
2	*A*	*Adjust Priority* *current Priority order: D > A > B* *new priority order : D = A > B*	*70.004* *28.011*	*1.000* *0.000*	*3269017*
3	*A*	*Adjust Priority* *current Priority order: D = A > B* *new priority order : A > D > B*	*72.41* *32.61*	*1.000* *0.000*	*3269017*
4	*B*	*Adjust Priority* *current priority order : A > D > B* *new priority order : A > D = B*	*72.34* *34.52*	*1.000* *0.000*	*3269017*
5	*B*	*Adjust Priority* *current priority order : A > D = B* *new priority order : A > B > D*	*72.34* *41.26*	*1.000* *0.000*	*3269017*
6	*C*	*Adjust Priority* *current priority order : E > C* *new priority order : E = C*	*62.00* *69.14*	*1.000* *0.001*	*3269018*
7	*C*	*Adjust Priority* *current priority order : E = C* *new priority order : C > E*	*62.00* *69.14*	*1.000* *0.001*	*3269018*
8	*C*	*Adjust Priority* *Candidate task has highest priority* *===> RESHAPE design* *1) reallocating candidate task* *if Candidate is reallocated to CPU1, new utilization will exceed 0.95.* *===> try split candidate task* *2) splitting candidate task* *current priority order : C > E* *new priority order : C_1 > C_2 > E*	*47.00* *69.14*	*0.000* *0.001*	*1.013*
9	*E*	*Adjust Priority* *new priority order : C_1 > C_2 >E* *new priority order : C_1 > C_2 = E*	*47.00* *69.14*	*0.000* *0.001*	*1.013*
10	*E*	*Adjust Priority* *new priority order : C_1 > C_2 = E* *new priority order : C_1 > E > C_2*	*47.00* *57.00*	*0.000* *0.000*	*0.000*

Table 5.2 Optimization steps of Experiment #2

step	Cand. task	Action	Resp. times to stim1 & stim2	Prob. of missing deadline	Solution penalty metric Eqn (5.3)
0			*70.00* *30.00*	*1.000* *0.000*	*3269017*
1	*A*	*Adjust Priority* *current Priority order: D > A = B* *new priority order : D > A > B*	*70.00* *30.00*	*1.000* *0.000*	*3269017*
2	*A*	*Adjust Priority* *current Priority order: D > A > B* *new priority order : D = A > B*	*71.36* *30.30*	*1.000* *0.000*	*3269017*
3	*C*	*Adjust Priority* *current priority order : E > C* *new priority order : E = C*	*70.02* *36.54*	*1.000* *0.000*	*3269017*
4	*C*	*Adjust Priority* *current priority order : E = C* *new priority order : C > E*	*77.50* *77.51*	*1.000* *1.000*	*6538034*
5	*E*	*Adjust Priority* *current priority order : C > E* *new priority order : E = C* *Current priority order has been visited befor ===> Adjust priority* *new priority order : E > C* *Current priority order has been visited befor ===> Adjust priority* *Candidate task has highest priority* *===> RESHAPE design* *reallocate candidate task to CPU1* *new priority order: E = D = A > B*	*70.02* *40.00*	*1.000* *0.000*	*3269017*
6	*A*	*Adjust Priority* *current Priority order: E = D = A > B* *new priority order : A > E = D > B*	*70.00* *41.00*	*1.000* *0.000*	*3269017*
7	*B*	*Adjust Priority* *current Priority order: A > E = D > B* *new priority order : A > E = D = B*	*70.00* *56.00*	*1.000* *0.000*	*3269017*
8	*B*	*Adjust Priority* *current Priority order: A > E = D = B* *new priority order : A > B > E = D*	*59.00* *57.00*	*0.000* *0.000*	*0*

Table 5.3 Optimization steps of Experiment #3

Chapter 6: Evaluation of the Framework

This chapter evaluates the framework developed in the thesis, in three ways:

1) Its applicability to real systems is tested by an industrial example. Section 1 describes the use of the framework to model and evaluate the performance of a real-life system: Automatic Protection Switching in a 2-fiber Bidirectional Line Switching Ring (BLSR) network [Bellcore95].
2) Its effectiveness, compared to other recently published and completely different approaches. Section 2 describes its application to two examples from the literature.
3) Its effectiveness over a large number of randomly generated problems of varying degree of difficulty. Section 3 describes how these problems were generated, and the results of the optimization method applied to 3000 systems.

6.1 The Automatic Protection Switching Case Study

In this case study, the framework was used to model and evaluate the performance of an automatic protection switching scenario in a 2-fiber BLSR network. The network has up to 16 nodes connected in a ring with each pair of adjacent nodes connected by a pair of fibers (one in each direction). Each node in the network consists of a pair of subnodes, corresponding to the two sides of the node. A 16-node 2-fiber BLSR network is shown in Figure 6.1

Protection capability, against network failure, is provided by having one of the subnodes being in standby mode while the other is the active traffic carrying subnode. When the active

subnode fails, the standby subnode will take over automatically as the active subnode and carry on the traffic.

For a single failure on a ring with less than 1200 KM of fiber, the protection operation is required to complete within 50 msec. An appropriate part of the 50 msec will be allocated to each node in the network based on the nature of the operations performed by the node during protection switching.

The next two subsections illustrate how protection switching is accomplished in response to a Loss of Signal (LOS) failure condition. The first subsection describes the high level behavior at the network level and shows the actions taken by each node. In the second subsection, a detailed sequence of actions on a node by node basis will be discussed.

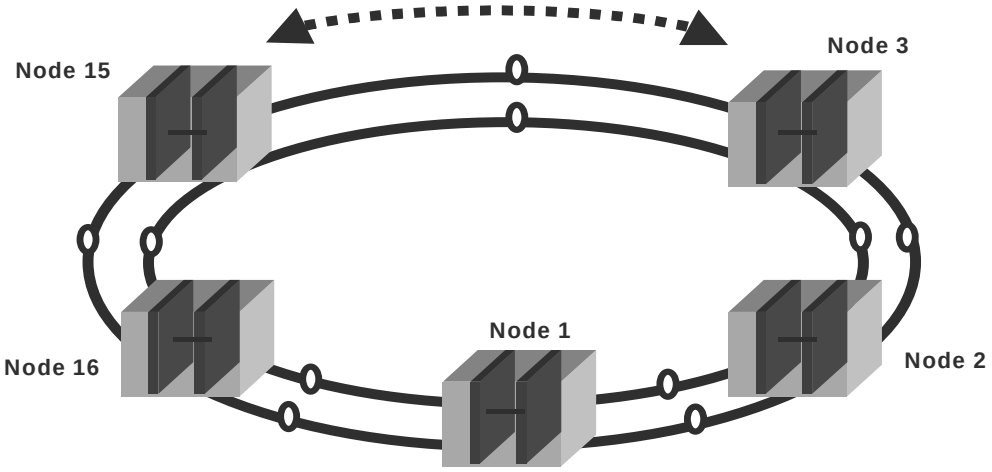


Figure 6.1 A 16-node 2-fiber BLSR network

6.1.1 Automatic Protection Switching at the network level

This section illustrates how protection switching reacts to a Loss Of Signal (LOS) failure condition. For the purpose of this discussion, the network is assumed to be idle and a uni-

directional fiber cut occurs on the fiber from node 16 to node 1. The failure is described as occurring on the fiber from node 16 to node 1, but the 31 other possible fiber failures are all symmetrical with this one and lead to essentially the same scenario, with just a node renumbering.

As soon as the failure occurs, the adjacent subnode on node 1 detects the failure as a LOS condition and initiates the following sequence of events:

Step1: Upon detecting the LOS failure, Node 1 sends a protection request message to Node 16 over the short and long path and enters the switching state. The short path request is sent directly to node 16, where as the long path request is sent to node 16 through node 2, 3,.... and 15.

Step 2: Upon receiving the short path request, node 16 sends a long path request indirectly to node 1 through node 15, 14,.... and 2.

Step 3: As the long path request sent by node 1 in Step 1 is traversing node 2 through 15, each of these nodes sees that the request is not addressed to it. So, each node immediately acts as an intermediate node, enters full-passthrough state and forwards the request on to its destination

Step 4: Similarly, as the long path request sent by node 16 in Step 2 is traversing node 15 through 2, each of these node will become intermediate node.

Step 5: Upon receiving the long path request sent by node 1, node 16 executes the bridge and switch operations and updates the protection status field in its K1 and K2 bytes (The K1 and K2 bytes are included in every SONET frame and used to transport protection switch requests to the appropriate nodes on the ring).

Step 6: Upon receiving the long-path request, node 1 performs the traffic bridge and protection switch operation and updates its K1 and K2 bytes.

Protection Switching is now complete.

The result of the protection switching operation is that the failed fiber segment, between node 1 and node 16, has been isolated and the traffic has been switched away from that.

6.1.2 Automatic Protection Switching on a node by node level

Figure 6.2 shows the SDL processes that participate in the automatic protection switching

scenario, as well as the way they interact. A brief description of the role of each process is given below.

- Fault Handler (FH): responsible for detection and handling of faults.
- Protection FSM (PFSM): responsible for implementing the BLSR protection protocol.
- Protection Controller (CTRL)
- Switching Manager (SM): responsible for control of the switch ASICs during normal and protection operations.
- Squelch Manager (SQM): responsible for traffic squelching to ignore further alarms.
- Auditing Manager (AM): responsible for traffic auditing.
- Mate Handler (MH): responsible for sending and receiving messages to and from the mate protection handler over the M2M link.
- Report Generator (RT)

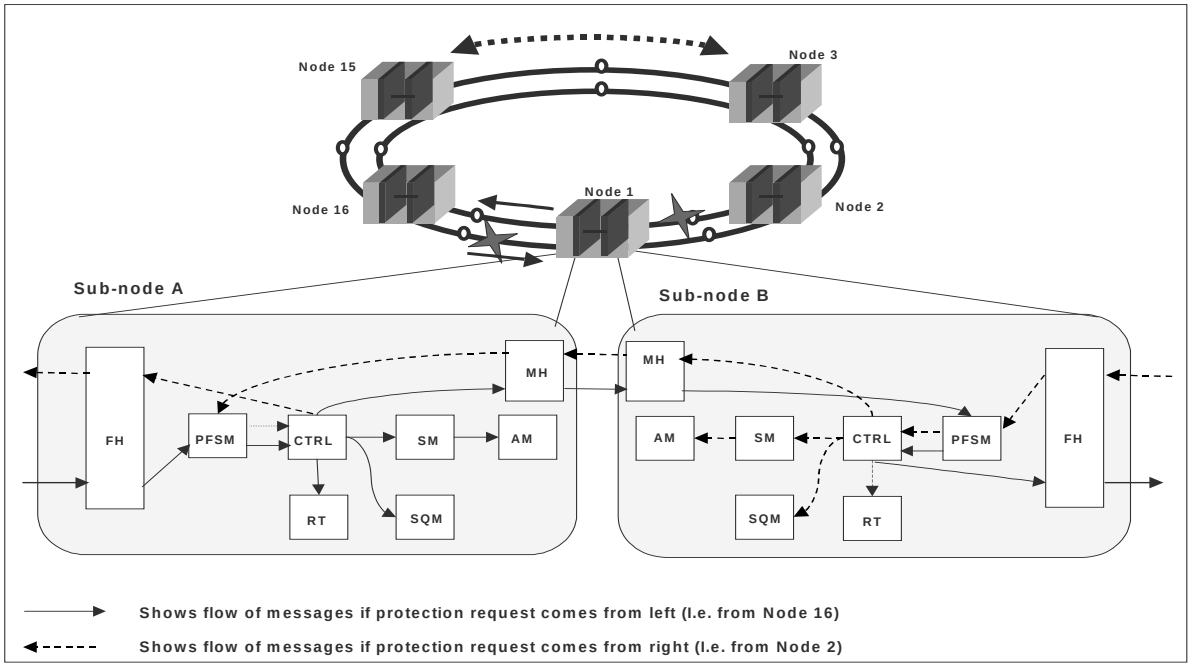


Figure 6.2 Software Architecture of The Automatic Protection Switching System

Depending on the type of message which a node receives during automatic protection switching, one of the following three scenarios can take place on the node.

1) The first sub-scenario describes the behavior of a node when it receives a LOS signal (For example, Node 1 in our case), or a “short_path” request (For example, Node 16 in our case). This scenario is shown in Figure 6.3.
2) The second scenario describes the behavior of an intermediate node (For example, Node 2 - Node 15 in our case), when it receives a “long path” request. This scenario is shown in Figure 6.4.
3) The last scenario describes the behavior of Node 1 and Node 16, when it receives a “long path” request. This scenario is depicted in Figure 6.5.

Scenario 1: Loss of signal scenario on node 1 after receiving the fault signal

Figure 6.3 describes the interactions among the processes and the sequence of actions that takes place on node 1, when it receives a fault signal. A similar scenario is also executed on node 16 when it receives the short_path request sent by node 1. A brief description of the scenario is given below.

Step 1: LOS is detected by the Fault Handler (FH_1_A) of node 1, subnode A.

Step 2: FH_1_A sends a Signal Failure Ring (SF-R) message to the Protection FSM process (PFSM_1_A).

Step 3: PFSM_1_A processes the message and sends the resulting actions to the protection Controller process (CTRL_1_A).

Step 4:

i) CTRL_1_A instructs the Mate Handler on the other subnode of node 1 (MH_1_B), over the M2M link, to take the actions required for the SF-R request.

ii) CTRL_1_A sends a “short path” request message to the Fault Handler (FH_1_A)

iii) CTRL _1_A instructs the Switch Manager (SM_1_A) to perform a traffic bridge operation.

iv) CTRL_1_A sends a message to the Squelch Manager (SQM_1_A) to replace traffic by the appropriate path Alarm Indication signal (AIS), to prevent misconnection in the BLSR ring configuration.

Step 5: FH_1_A forwards the “short_path” request message to process FH_16_B on Node 16.

Step 6: SQM_1_A inserts the appropriate path AIS instead of the traffic destined to the failed node and/or the traffic expected from the failed node.

Step 7: SM_1_A updates the switching hardware with the Bridge map and sends a message to the Audit Manager (AM_1_A).

Step 8: At Subnode B, the Mate Handler (MH_1_B) receives a message from its mate (MH_1_A in step 4 above) and forwards it to PFSM_1_B.

Step 9: PFSM_1_B sends a message to CTRL_1_B

Step 10: CTRL _1_B sends a “long_path” request (destined to Node 16) to FH_1_B

Step 11: FH_1_B forwards the “long_path” request to next node (Node 2, process FH_2_A)

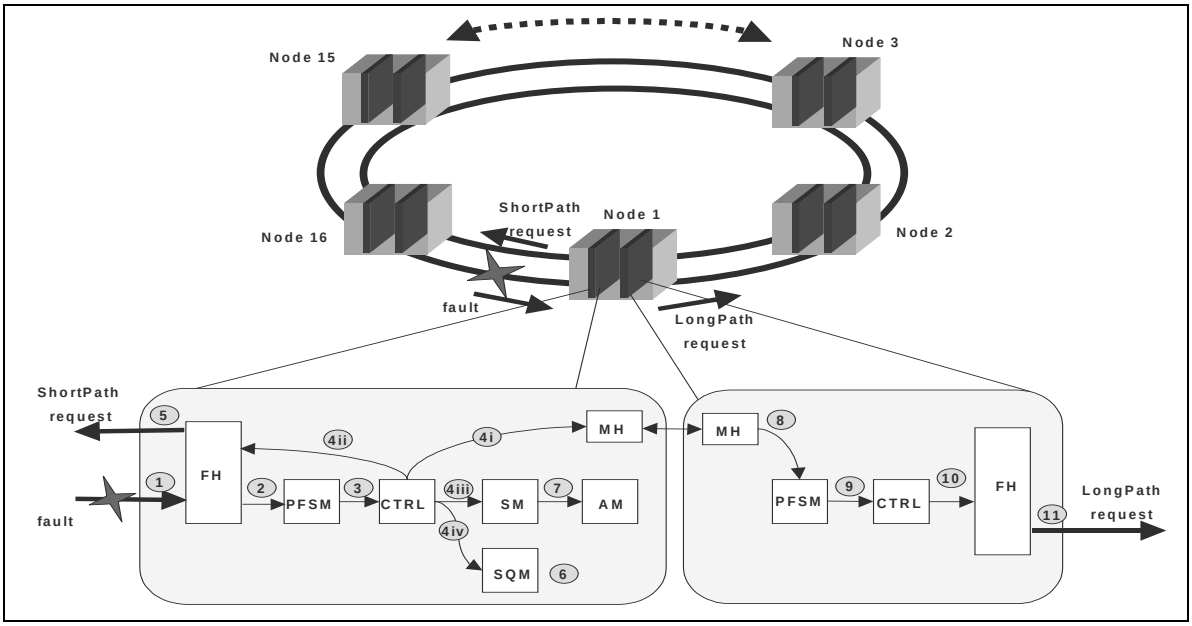


Figure 6.3 Scenario1: Sequence of actions that take place at Node 1 when faults occurs

Scenario 2: Loss scenario on an intermediate node (node 2 - node 15).

The sequence of actions that take place on an intermediate node in response to receiving a “long_path” request from its neighbor node is shown in Figure 6.4. This scenario is explained below in the context of node 2 when it receives a “long_path” request from node 1.

Step 1: Fault Handler (FH_2_A) receives a “long_path” request message from Node 1 and forwards it to PFSM_2_A.

Step 2: PFSM_2_A processes the message and sends the resulting actions to the protection Controller process (CTRL_2_A).

Step 3:

i) CTRL_2_A instruct the mate handler on the other subnode (MH_2_B), over the M2M link, to take the actions required for the SF-R request.

ii) CTRL_2_A sends a message to the Squelch Manager (SQM_2_A) to replace traffic by the appropriate path AIS to prevent misconnection in the BLSR ring configuration.

iii) CTRL_2_A instructs the Switch Manager to perform a traffic paththrough operation.

iv) CTRL_2_A sends a “protection_status_update” message to Report Generator (RT)

Step 4: SQM_2_A inserts the appropriate path AIS instead of the traffic destined to the failed node and/or the traffic expected from the failed node.

Step 5: SM_2_A update the switching hardware with the appropriate map and sends a message to the Audit Manager (AM_2_A)

Step 6: At Subnode B, the mate handler (MH_2_B) receives a message from its mate (MH_2_A in step 3 above) and forwards it to PFSM_2_B.

Step 7: PFSM_2_B sends a message to CTRL_2_B

Step 8: CTRL_2_B sends a “long_path” request message to FH_2_B

Step 9: FH_2_B forwards the “long_path” request to next node (i.e. Node 3 process FH_3_A)

Scenario 3: Loss scenario after receiving the long_path request

The sequence of actions that take place on Node 1 when it receives the long_path request sent by node 16 is shown in Figure 6.5, and is explained below. A similar scenario is also executed on node 16 when it receives the long_path_request sent by node 1.

Step 1: Fault Handler (FH_1_B) receives a “long_path” request from Node 2 (process FH_2_A) and forwards it to PFSM_1_B.

Step 2: PFSM_1_B processes the message and sends the resulting actions to the protection Controller process (CTRL_1_B).

Step 3:

i) CTRL_1_B sends a message to the Squelch Manager (SQM_1_B) to replace traffic by the appropriate path to prevent misconnection in the BLSR ring configuration.

ii) CTRL_1_B instructs the Switch Manager to perform a traffic switch operation.

Step 4: SQM_1_B inserts the appropriate path AIS instead of the traffic destined to the failed node and/or the traffic expected from the failed node.

Step 5:

i) SM_1_B update the switching hardware with the appropriate map. ***Protection is now completed and signalling reaches steady state.***

ii) SM_1_B sends a message to the Audit Manager (AM_1_B)

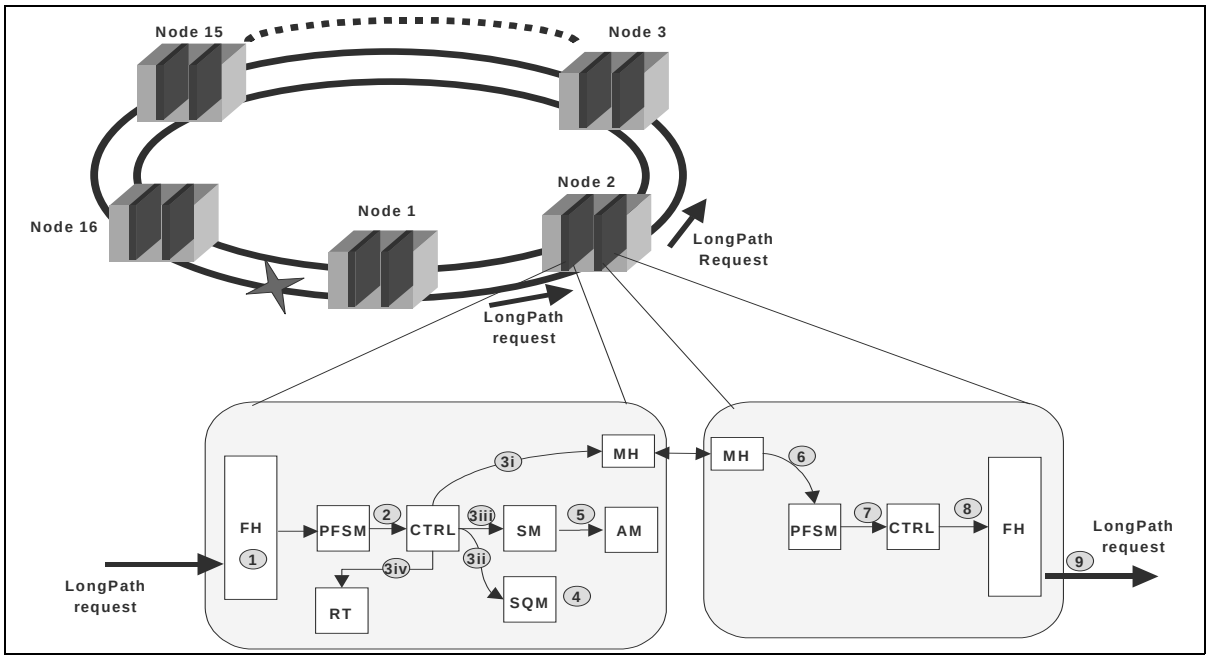


Figure 6.4 Scenario 2: Sequence of actions that take place at an intermediate node

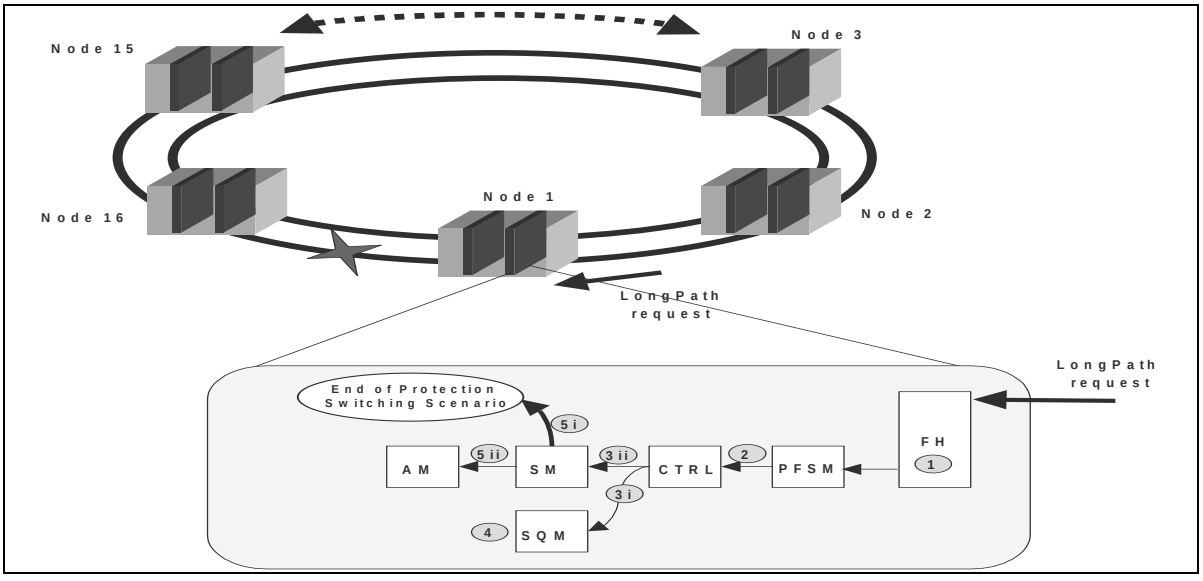


Figure 6.5 Scenario 3: Sequence of actions that take place at Node 1 after receiving the long_path request from Node 16

6.1.3 Modeling and Evaluation

In the previous sections, the various scenarios that took place during automatic protection switching were described. Using the proposed Performance Engineering Framework, these scenarios were captured in the Scenario Language (SL), then used to build a LQN model, as was described in Chapter 4.

Figure 6.6 shows the generated LQN model. If you compare the diagram to the scenarios in Section 6.1.2, you can verify that it has captured all the processes that participated in the system as well as their interactions. It also captures the end-to-end behavior of the automatic protection switching scenario. For example, once a fault is detected by the FH process on Node 1, Subnode A, a signal is sent to FSM process which in turn forwarded it to the CTRL process. After the CTRL process has processed the fault message, it directly informs its neighboring node (which is Node 16 in this case) about the fault, and initiates the propagation of the fault message, indirectly to Node 16 through the network, via the mate handler (MH) process. Once the message sent to Node 16 is propagated back to Node 1, through Node 15-2, and processed by the switching manager process on Subnode-B of Node 1, the scenario is said to be complete. This end-to-end behavior is captured in the model by the chain of Forwarding messages, which are shown as dashed arrows. The end of the forwarding chain is shown in the figure by a heavy arrow attached to the process SM of Subnode B, Node 1, indicating a reply to the fault generator and the end of the response.

The parameters of the model were determined by the demand and behavior shown in Table 6.1. Process priorities on each subnode have been assigned as follow (where MH stands for the priority of process MH): MH = FH > PFSM = CTRL > SM = SQM > AM. The model was solved using the LQN solver and the response time was found to be 47.16 msec, which is compliant with its requirement of 50 msec.

To test the optimization strategy, the model was made unfeasible by lowering the priority of the FH process (on subnode A, node 1) 2 levels, then the model was solved again. The results are given as step 0 in Table 6.2, and show a response time of 63.2 msec, which is non-compliant because it exceeds the requirements of 50 msec. The optimization was applied to the non-compliant design and managed to find a feasible solution in 3 steps as shown in Steps 1 to 3, in Table 6.2, using only changes in priority.

Being able to automatically capture the structure and behavior of the Automatic Protection Switching system in an LQN model, this demonstrates that the framework can produce a model which accurately represent the original scenario, automatically without user intervention. By being able to transform the infeasible design to a feasible one in only 3 steps, this shows that the proposed framework is robust and able to find feasible solutions efficiently.

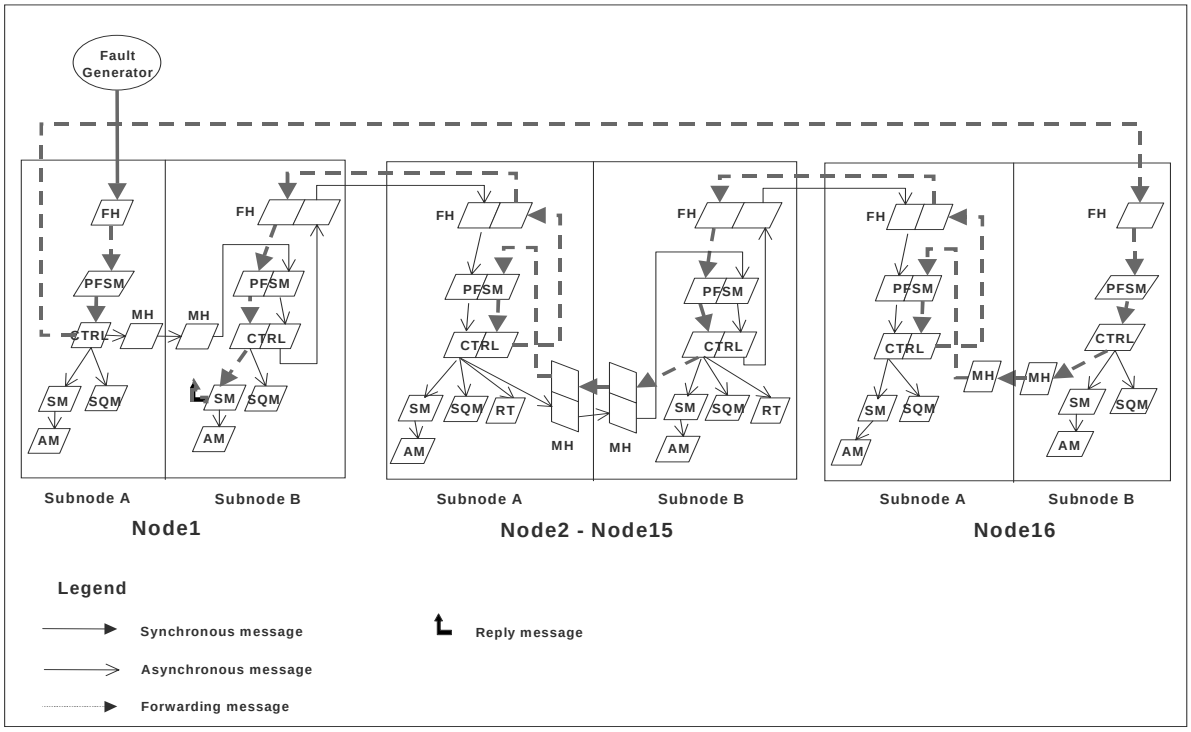


Figure 6.6 LQN model of the automatic protection switching system

Process	Sequence of activities and their service times
FH	• executes for 0.1 msec
CTRL	• executes for 0.1 msec. • sends two messages to FH, MH • executes for 1 msec • sends two messages to SQM and SM • on an intermediate node, it further executes for 4 msec then sends one message to RT
PFSM	• executes for 1.5 msec., when it receives a message from FH. However, it executes for 0.02 msec when it receives a message from MH.
MH	• executes for 0.1 msec.
SM	• executes for 10 msec.
SQM	• executes for 2 msec.
AM	• executes for 100 msec.
RT	• executes for 20 msec.

Table 6.1 Sequence of activities and service times associated with each process in the protection switching case study

Step	Candidate process	Action	Response time (msec.)	Prob. of missing deadline
0			*63.2*	*1.0*
1	*FH_1_A*	*Adjust Priority* *Current priority order:* *MH_1_A > PFSM_1_A = CTRL_1_A > SQM_1_A = SM_1_A = FH_1_A > AM_1_A* *New priority order:* *MH_1_A > PFSM_1_A = CTRL_1_A > FH_1_A> SQM_1_A = SM_1_A > AM_1_A*	*63.2*	*1.0*
2	*FH_1_A*	*Adjust Priority* *Current priority order:* *MH_1_A > PFSM_1_A = CTRL_1_A > FH_1_A > SQM_1_A = SM_1_A > AM_1_A* *New priority order:* *MH_1_A > PFSM_1_A = CTRL_1_A = FH_1_A > SQM_1_A = SM_1_A > AM_1_A*	*52.36*	*1.0*
3	*FH_1_A*	*Adjust Priority* *Current priority order:* *MH_1_A > PFSM_1_A = CTRL_1_A = FH_1_A> SQM_1_A = SM_1_A = > AM_1_A* *New priority order:* *MH_1_A > FH_1_A > PFSM_1_A = CTRL_1_A > SQM_1_A = SM_1_A > AM_1_A*	*47.16*	*0*

Table 6.2 The optimization steps of the protection switching case study

6.2 Comparing with other methods

The purpose of this section is to compare the optimization approach with other approaches in the literature. Since we do not know of any heuristic algorithm that finds a feasible process allocation, structuring and priority assignment for the type of software architectures and designs addressed in this research (i.e. which support servers with multi-services and various communication paradigms), straight comparison with other heuristics is difficult. However, since Tindell et al. [Tindell92], Santos et. al [Santos97] and Etemadi [Etemadi96] proposed several heuristic methods to solve a problem similar to the one treated here, they can be used as a reference mark.

6.2.1 Tindell's Example

Tindel et. al. [Tindell92] and Santos et. al. [Santos97] proposed two heuristic algorithms for solving the problem of process allocation and priority assignment combined. Figure 6.7 Shows the system of 11 real-time periodic transactions and 43 activities used as illustrative example in [Tindell92], and then in [Santos97]. The figure also shows the period, worst case execution time, communication load and, if it is defined, the pre-allocated processor for each activity. The number within square brackets in an activity represents the execution time of the activity. The number within a square besides an activity indicates the processor to which the activity is preallocated. The number besides an arrow indicates the size of message in bytes.

To model this system using our framework, first the transactions were represented as scenarios, using the Scenario Language, with every activity mapped to a separate process. Then the corresponding SDL model was generated and executed. Finally the LQN model was extracted (as was described in Chapter 4), and the optimization strategy was applied (as was described in Chapter 5).

For comparison purposes, the optimization method was first applied on the original problem reported in both papers, which uses 8 CPUs and has 1 msec. for propagation delay. Then, to show the flexibility and superiority of our approach, the complexity of the problem was increased in different ways and four more experiments were conducted to solve the more complex problems.

Table 6.3 summarizes the results of all experiments. In each experiment, the problem was solved 8 times, each time a different allocation strategy in MULTIFIT-COM was used, as described in Section 5.2, to give a new initial solution to start with. Success ratio is defined here as the fraction of the eight attempts which found a feasible solution.

- The first experiment (Experiment # 1) shows the results of the original problem with 8 CPUs, along with the success ratio and average number of steps taken by the optimization method to find a feasible solution.
- In Experiment #2, a computation overhead of 0.001 msec/byte was included each time a message was sent or received. Although this overhead causes an increase of 15% in the execution demand of some processes, it does not affect the success ratio of the optimization method. Only the average number of steps to find a feasible solution was increased.
- In Experiment #3, a computation overhead of 0.003 msec/byte was included each time a message was sent or received. This overhead causes an increase of 30% in the execution demand of some processes; however, it only degraded the success ratio from 8/8 to 7/8.
- In Experiment # 4, the number of CPUs was reduced to 7 CPUs but the overhead due to message sending and reception was not considered. This new configuration reduced the success ratio to 6/8.
- In Experiment #5 with 7 CPUs, an overhead of 0.001 msec/byte sent and received was included and this further reduced the success ratio to 5/8.

For Tindell and Santos's problem (Experiment #1), the present method was as good as theirs,

that is it also found feasible solutions for all 8 starting points. For more stressful problems, with communication overhead costs and/or less number of CPUs (Experiments #2 to #5), the present method succeeded for most but not all starting points.

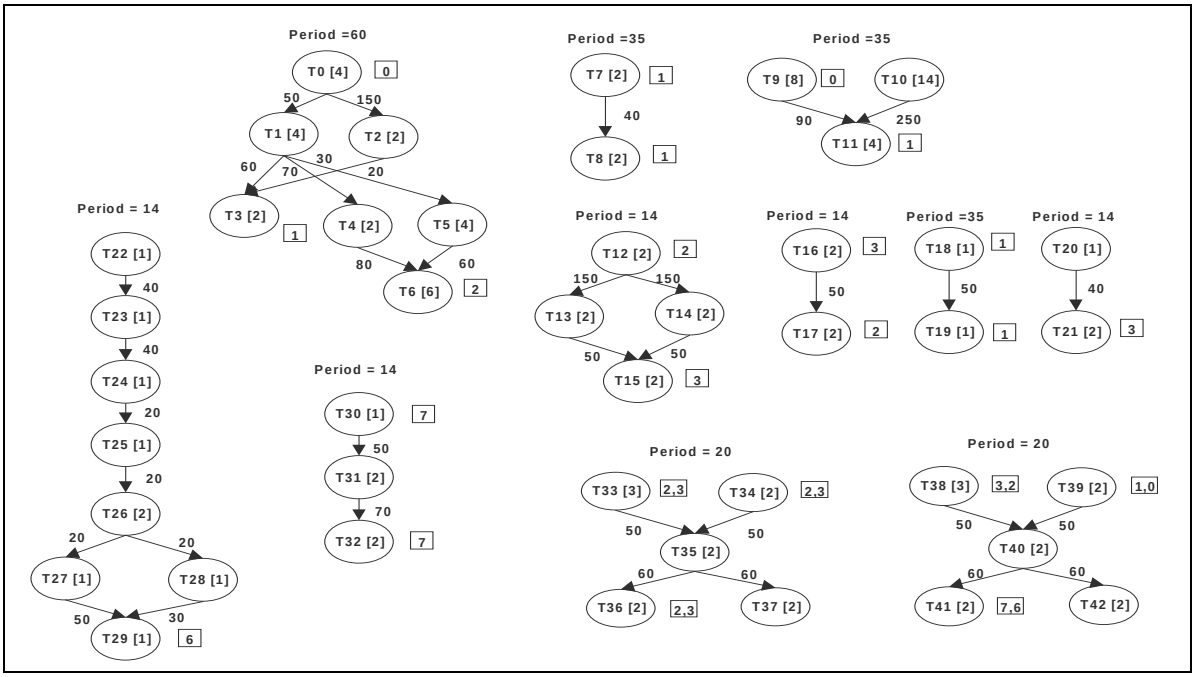


Figure 6.7 The 11 transactions of Tindell's example

Exp. #	No. of CPUs	Propagation Delay (msec)	Overhead per byte sent or received	Success Ratio	Avg. No. Of steps for successful cases
1 (original)	***8***	***1***	***0***	***8/8***	***4***
2	***8***	***1***	***0.001***	***8/8***	***16***
3	***8***	***1***	***0.003***	***7/8***	***23***
4	***7***	***1***	***0***	***6/8***	***26***
5	***7***	***1***	***0.001***	***5/8***	***51***

Table 6.3 Optimization experiments using Tindell's example

6.2.2 Etemadi's Example [Etemadi96]

In Etemadi's Ph.D. thesis, three heuristic algorithms were proposed to calculate process priorities for a given process allocation. The algorithms were applied to the sonar signal processing component of a towed array sonar system with 12 transactions, shown in Figure 6.8. In this figure, the number within square brackets in an activity represents the execution time of the activity. The number within a square besides an activity indicates the processor to which the activity is preallocated. The number besides an arrow indicates the size of message in bytes.

In the basic configuration, where the system has 12 transactions, all three algorithms succeeded to find a feasible priority assignmnet. However, when a second copy of transaction 1, with a period of 1125 time units, was added to the basic configuration, only two of Etemadi's algorithms succeeded to find a feasible priority assignment.

To model this system using our framework, first the transactions were represented as scenarios, using the Scenario Language, with every activity mapped to a separate process. Then the corresponding SDL model was generated and executed. Finally the LQN model was extracted (as was described in Chapter 4), and the optimization strategy was applied (as was described in Chapter 5).

To compare the optimization method with the algorithms proposed in [Etemadi96], the present method was applied to his extended sonar system which has two copies of transaction 1. The method managed to find a feasible priority assignment in the initialization stage without performing any optimization steps. Two experiments were then conducted on a more challenging version of the problem, with fewer CPUs, after removing the original preallocation constraints. The results of these experiments are summarized in Table 6.4. In each experiment, the problem was solved 8 times, each time a different allocation strategy in MULTIFIT-COM

was used, as described in Section 5.2, to give a new initial solution to start with. Success ratio is defined here as the fraction of the eight attempts which found a feasible solution.

- Experiment #1 shows that for 8 CPUs the initial allocation is feasible.
- In Experiment #2, the number of CPUs was reduced to 7. Although, this reduction in available resources does not affect the success ratio, two steps were now required to find a feasible solution.
- In Experiment #3, the number of available CPUs was reduced to 6. However, the optimization method managed to find many feasible solutions with a success ratio of 7/8.

In conclusion, the proposed method handled the original examples successfully. Also, after the examples were made more difficult, it still succeeded in finding feasible solutions.

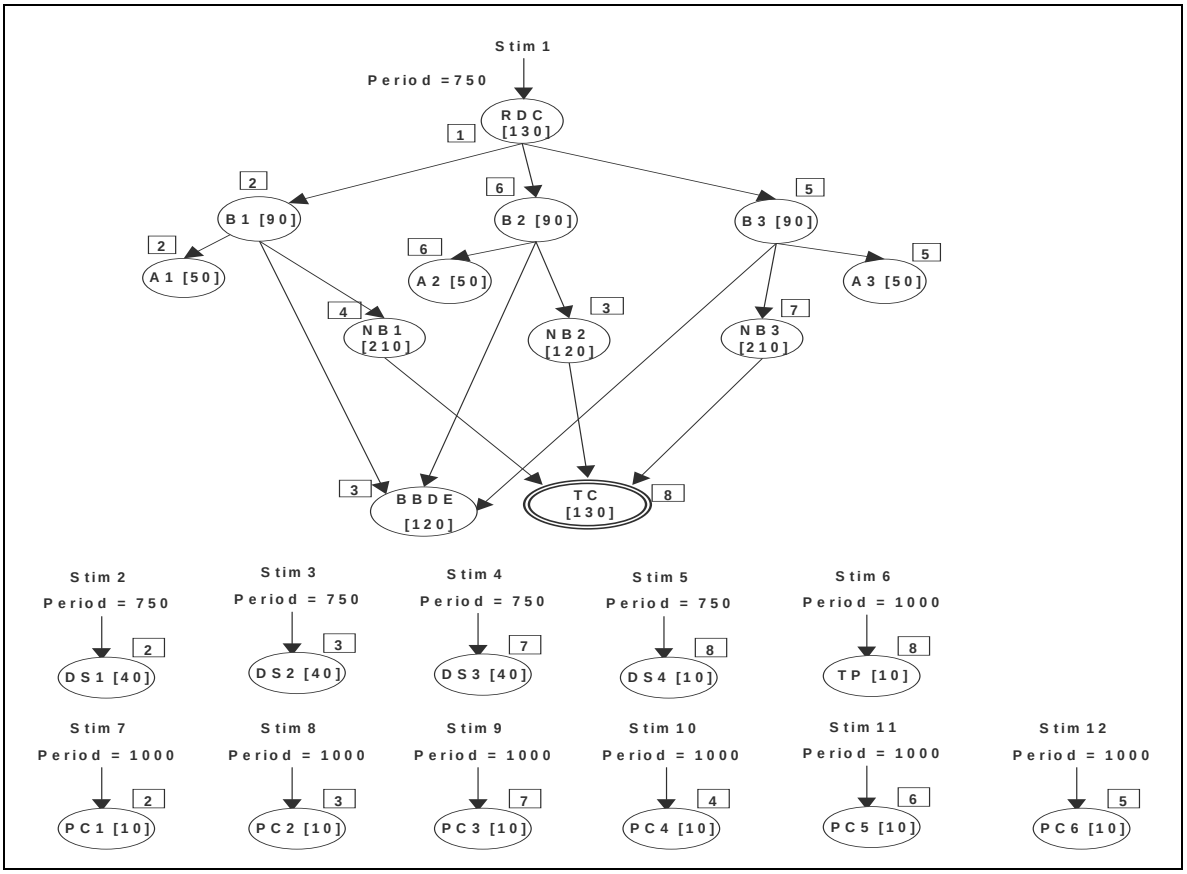


Figure 6.8 The 12 transactions of Etemadi's example

Exp. #	No. of CPUs	Success Ratio	Avg. No. Of steps for successful cases
1	*8*	*8/8*	*0*
2	*7*	*8/8*	*2*
3	*6*	*7/8*	*60*

Table 6.4 Optimization experiments using Etemadi's example

6.3 The Optimization Method: Robustness on Random Examples

This section tests the optimization method of Chapter 5 on a wide variety of applications which were generated as LQNs with 16 processes and a randomly generated pattern of interaction and execution.

There are a large number of aspects that may affect the performance of the optimization strategy. They can be classified as system aspects which specify the distributed system under consideration and application aspects which specify the software architecture and the communication patterns between processes. The generation of realistic applications and distributed systems largely depend on how these aspects are specified. However, little is reported in the literature about “typical” real-time software architecture and their communication patterns. Thus, in our experiments, we randomly generate both the software architecture and process parameters according to some guidelines as will be discussed below. We believe that these randomly generated software designs cover a wide spectrum of real-time applications.

To assess the performance of the optimization method, it was applied to 3000 randomly generated software models allocated on 4 CPUs. The models were generated by assigning random execution times to the entries of the two software architectures shown in Figures 6.9 and 6.10. For each software architecture, 1500 software models were randomly generated. Each of these has the following characteristics:

- Number of scenarios = 4
- Number of processes = 16
- Probability that a process has more than one entry = 0.6

- Probability that a process participate in more than one scenario = 0.6
- Average number of messages sent by a process = 1.3 message.
- Communication overheads and delays are set to zero (ignored).
- The computation time of an entry is uniformly distributed between [80,120] time units.
- The period of a scenario is set according to equation (6.1)
- The deadline of a scenario is set according to equation (6.2)

$$Period_s = C_e \times N_c \times F \quad (6.1)$$

$$Deadline_s = C_e \times N_c \times L \quad (6.2)$$

C_e = Average cost (computation time) of an entry

N_c = Number of entries along the critical path of scenario S

F = Period flexibility factor

L = Laxity factor, ranging from 2.1 to 3.0

Under the above formulas, given a scenario with certain computation requirements, the larger the period flexibility factor (F), the larger the period that will be chosen for scenarios, and hence the less intense the workload that will be generated by them. Accordingly, the period flexibility factor was used during the experiments to change the average workload intensity and the average utilization of the CPUs. It was chosen to provide average CPU utilization between 0.4 and 0.8. On the other hand, the larger the laxity factor (L), the larger the deadline that will be assigned to scenarios, hence the easier is it to find a feasible design. Thus, the Laxity Factor was used during experiments to control the laxity of scenarios, and was chosen to range from 2.1 to 3.0.

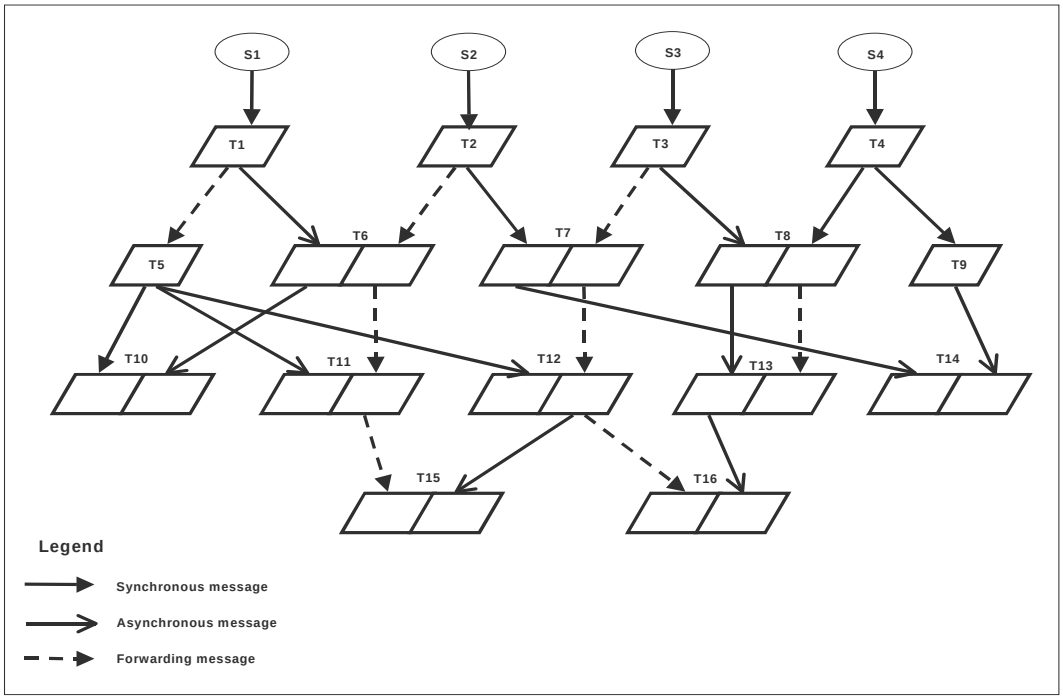


Figure 6.9 The software Architecture of the random application1

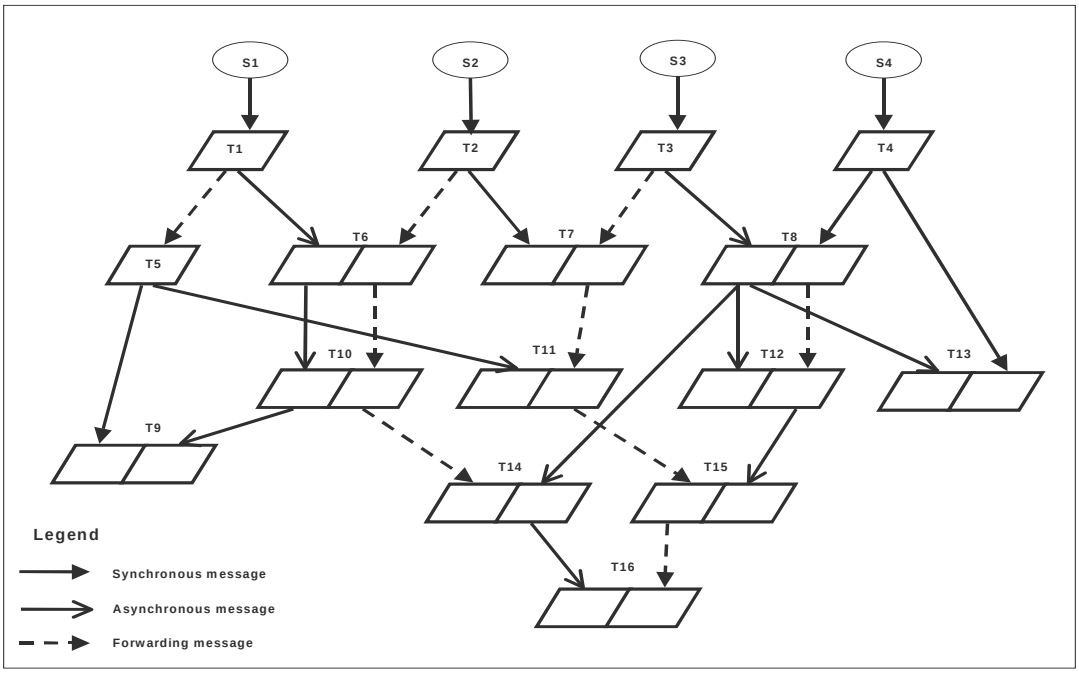


Figure 6.10 The software Architecture of the random application2

Before applying the optimization method to a model, we removed from consideration the models that had infeasible execution requirements. This was detected by calculating the minimal time requirements of a scenario, which is the total execution time along the critical path of the scenario, and comparing it with the deadline of the scenario. If the minimal time requirements of any scenario was greater than its deadline, the design was identified as “obviously infeasible” and not considered any further. Obviously, this does not eliminate all infeasible designs since the interference and contention with other processes, which are not on the critical path, can make some additional designs infeasible. However, the problem of determining if a given design is feasibly schedulable in multiple processor is a computationaly intractable problem [Ramamritham95]. Because of this, when a heuristic algorithm does not succeed in determining a feasible schedule, it could be due to the infeasibility of the design. Hence the metric chosen to characterize the performance of the algorithm is the Success Ratio. If an algorithm is able to find feasible designs for x out of the given y designs, (where none of the y designs are obviously infeasible) its Success Ratio (SR) is said to be (x/y).

The optimization method was applied to the randomly generated designs and its performance was determined for different average processor utilization factor (APUF) and laxity constraints, where APUF is the sum of the utilization factors of all processes divided by the number of processors. Results are presented in Figure 6.11 and Figure 6.12, and are summarized as follows:

- For a given Laxity Factor, the Success Ratio decreases as the APUF increases.
- For a given APUF, the Success Ratio increases as the Laxity Factor increases.

It can be seen from the results that the strategy and heuristics used for optimization is robust and guides the algorithm efficiently for a feasible design. For example, the optimization method succeeded in finding a feasible solution more than 80% of the time when the Laxity factor was

2.6 or greater, or the average CPU utilization was less than 50%. In addition, when the average CPU utilization was around 60%, the optimization method only performed 20 steps on average to find a feasible process allocation and priority assignment to the software design. Even when the average CPU utilization was as high as 80%, the optimization method only performed 35 steps on average to find a feasible solution.

Note that the purpose of the above experiments was not to completely quantify the behavior of the algorithm, but rather to verify its general performance trends. Thus, for examples, we conducted the experiments on problems based only on two software architectures. The conclusion we draw from the above experimental observation is that the optimization method is robust and has potential to find feasible solutions efficiently. In particular, the small number of steps relative to the size of the search space shows that the metric used to identify the contributors for performance problems is efficient. Also the high success ratio shows the robustness of the overall optimization method.

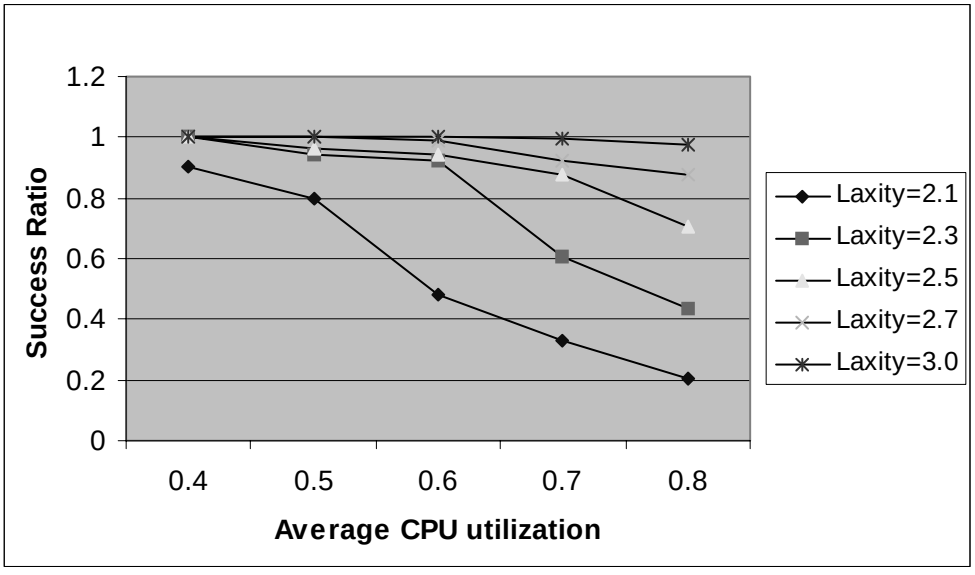


Figure 6.11 Optimization results: success ratio vs. average CPU utilization

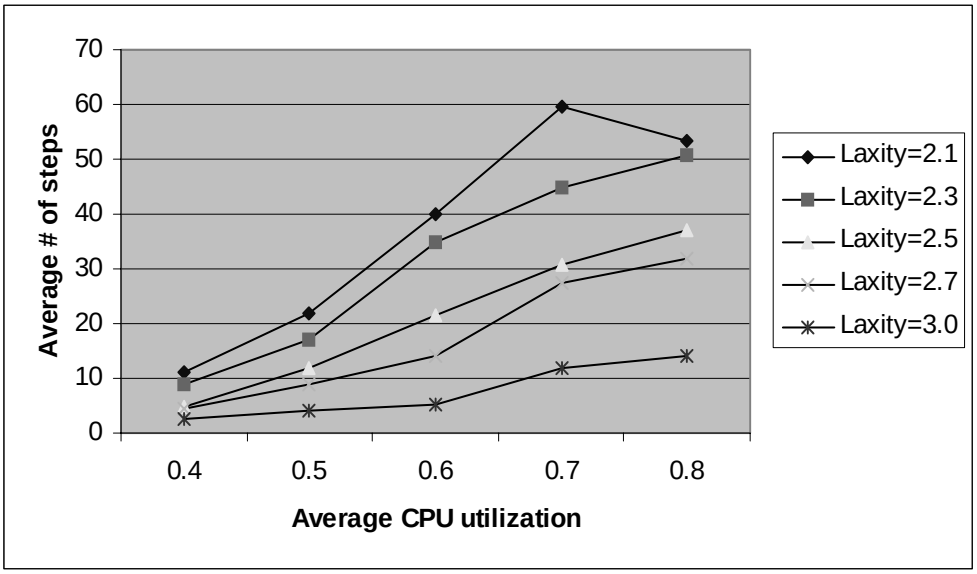


Figure 6.12 Optimization results: average # of steps vs. average CPU utilization

Chapter 7: Conclusions

7.1 Discussion

This thesis has described a complete process for analyzing performance of designs using models. One of the principal concepts of the process is to create the model *from the design*, so it can be forced to track the design closely as it evolves, and to introduce automation to make the tracking cheap and rapid. The starting point is a design (in an asynchronous style) expressed in SDL processes, which are constructed from a set of scenarios. To build a performance model, the SDL model is executed for a set scenarios, traces are recorded for each scenario, and the model structure and data is extracted from the traces. A layered queueing model is then constructed. Model building is completely automatic.

For asynchronous designs that are encoded in SDL, the model builder is operational and has been tested on a large number of designs created as test cases, with different features and degrees of complexity. The use of the model-builder has also been demonstrated on a large example (Automatic Protection Switching), with 252 processes.

The performance models produced in this work reflect (by construction) the structure of the software or system design in SDL, combined with the coverage of the scenarios. If some part of the design is not activated by the scenarios then it will not be in the model, but the omission may be deliberate. For example, exception-handling code that is rarely active will not affect the normal performance of the application.

The question of accuracy of the final predictions is not addressed here. Our purpose is to

create a model containing everything that is in the scenarios and the SDL design. The question of model accuracy then becomes a question of coverage of the scenarios, adequacy of the configuration definitions, and accuracy of the resource functions. These are important questions but are beyond the scope of this thesis. In its present state the process depends on engineering judgement to give adequate coverage and accuracy.

The performance models can be used for all the usual kinds of analysis. The accuracy of predictions will depend on the accuracy of the resource function data which must also be provided. Approximate values will give approximate results, which may be entirely adequate for early analysis. In our own work we often find that errors of less than 10% are excellent for this purpose, and errors less than 25% are adequate.

From the earlier work in [Hrishuck95], the model builder has retained the concept of angio traces and the goal of model building, but it has broadened the target from processes based on RPCs to a system (that is, SDL) based entirely on asynchronous messages, with additional types of execution paths (e.g. parallel execution) and interactions. To deal with the broader problem the model-building algorithms have been reconstructed from the ground up, beginning with the trace capture and interpretation. Only the term “angio trace” and the outline of steps have been retained.

The model building technique can be applied without using SDL; other early design descriptions, such as Use Cases or prototypes, can in principle be used to provide inputs to the same model-building process. In [Hrischuk95] another CASE tool, ObjecTime [ObjectTime97], was the source of angio traces. Even target code could be instrumented and traced to build models, and this would let us maintain the same models from early in the design cycle, into production code.

The goal of optimization is to satisfy soft deadlines in responses. The optimization strategy proposed here improves a non-compliant design by making incremental changes. The approach taken is to identify the factors that might contribute to the problem, rank them, and from the ranking generate a design change. The new design is then evaluated and the quality of the solution is estimated. The technique iterates between these steps until either a feasible solution is found or some stopping criteria are satisfied.

The optimization process is formulated as a two-level optimization problem. At the first-level of optimization, only changes in process priorities are considered as candidate factors until the solution converges without obtaining feasibility. The second-level of optimization reshapes the design by changing the process allocation and/or structure, and the search is restarted again at the first-level.

One of the unique features of the optimization strategy is that it can be applied on complex software architectures (i.e. which include servers with multi-services and various communication paradigms). To our knowledge, this class of software architectures can not be handled by any optimization algorithm in the literature.

Another important advantage of the optimization strategy is that it uses an incremental approach for identifying and solving performance problems due to wrong design decisions on priority assignments, process allocation and/or process structuring. This means that the optimization method can be applied to an advanced design (in which its configuration parameters are obtained by performing slight changes to an original feasible design, as in the case of the Automatic Protection Switching example), as well as to initial designs (in which its configuration parameters are obtained by applying some heuristic techniques, as in the case of the two examples from literature and the randomly generated examples).

In addition, the optimization strategy can solve performance problems due to one design decision (i.e. priority assignments, process allocation and/or process structuring) or to all the three decisions combined. In the literature, most approaches start from initial designs, and develop optimal (or near optimal) solutions to each subproblem (design decision) separately. Unfortunately, it is often not possible to obtain good solutions (or even feasible solutions) if the three design decisions are treated in isolation [Tindel92].

The optimization strategy has been used to optimize the performance of an industrial design: Automatic Protection Switching in a 2-fiber Bidirectional Line Switching Ring network. In this case study, the original feasible design has been slightly altered and made unfeasible by changing the priority of some process, and was found that the optimization method succeeded to transformed the infeasible design to a feasible one in only 3 steps. This shows that the proposed method is robust and able to find feasible solutions efficiently.

The optimization strategy has been compared with other approaches in the literature. It has been applied to the example proposed by Tindell et al. [Tindell92] and Santos et. al [Santos97], as well as to the example proposed by Etemadi [Etemadi96]. For Tindell and Santos's problem, the present optimization method was as good as theirs, that is it also found feasible solutions for different starting points. For more stressful versions of their problem, with communication overhead costs, the present method succeeded for most but not all starting points. Similar results were observed with Etemadi's problem. In conclusion, the proposed method handled the original examples successfully. Also, after the examples were made more difficult, it still succeeded in finding feasible solutions.

The optimization method was also applied to 3000 randomly generated designs and its performance was determined for different average processor utilization factors (APUFs) and laxity constraints, where APUF is the sum of the utilization factors of all processes divided by

the number of processors. It has been seen from the results that the strategy and heuristics used for optimization is robust and guides the algorithm efficiently for a feasible design. For example, the optimization method succeeded to find a feasible solution more than 80% of the time when the Laxity factor was 2.5 or greater, or the CPU utilization was less than 50%. In addition, when the average CPU utilization was around 60%, the optimization method only performed 20 steps on average to find a feasible process allocation and priority assignment to the software design. Even when the average CPU utilization was as high as 80%, the optimization method only performed 35 steps on average to find a feasible solution.

The conclusion we draw from the above experimental observations is that the optimization method is robust and has potential to find feasible solutions efficiently. In particular, the small number of steps relative to the size of the search space shows that the metric used to identify the contributors for performance problems is efficient. Also the high success ratio shows the robustness of the overall optimization method.

7.2 Summary of Contributions

The main contributions of this thesis are summarized below.

- A conceptual framework for performance engineering of real-time distributed systems is introduced (Chapter1).
- A way for specifying software designs with parallel threads that fork-and-join is explained (Chpater3).
- A new methodology and a tool for automating the construction of Layered Queueing Network models from the execution traces of SDL functional designs is proposed (Chapter 4).
- A new technique and a tool for obtaining compliance with response deadlines is proposed (Chapter 5).

- The framework has been implemented in a toolset by integrating all the tools that implement the individual components, namely the software specification tool (Scenario Design Paradigm), the SDL Tool, the LQN model builder tool, the LQN model solver tool, and the optimization tool. This toolset enables practitioners (architects/designers/developers) with little training to establish and maintain performance baselines for complex, real-time distributed systems.
- The framework has been evaluated in 3 ways (chapter 6):
  1) Its applicability to real systems is tested by an industrial example. The framework was used to model and analyze the performance of an Automatic Protection Switching system (with 252 processes), in a 2-fiber Bidirectional Line Switching Ring (BLSR) network.
  2) Its effectiveness, compared to another recently published and completely different approaches has been investigated. The framework has been applied to 2 examples from the literature, and was found that optimization strategy is as good as other techniques on standard problems.
  3) Its effectiveness over a large number of randomly generated problems of varying degree of difficulty has been demonstrated.

7.3 Future Work

In order to develop a complete toolset for performance engineering of real industry designs, some issues addressed in this thesis need to be extended. In particular, the following two aspects may have the potential for building upon the current work:

Increasing the practical usefulness of the framework

To increase the practical usefulness of the methods described here, it could be useful to:

- implement a more general interface for trace processing
- develop a technique/tools for deriving resource demands from system specification

- extend the trace and the model building algorithm for more general styles (i.e. software designs that are not necessarily asynchronous)

Improving the optimization Strategy

Although the optimization method presented in the thesis has been found satisfactory; it may be valuable to extend and improve it in the following directions.

- better diagnostic measures to identify and rank the sources of performance problems.
- better policy and heuristics for selecting the major source(s) for performance problems, and recommending configuration/design changes.
- new formulation for the optimization problem and solution. In this thesis, the optimization problem has been formulated as a two-level optimization problem, and an incremental approach has been proposed to solve it.

References

[Arora80] R.K. Arora and S.P. Rana. “Heuristic algorithms for process assignment in distributed computing system”, Information Processing Letters, 11(4,5):199-203, 1980.

[Audsley91] N.C. Audsley, “Optimal priority assignment and feasibility of static priority tasks with arbitrary start times”, Department of Computer Science, Report no. YCS 164, University of York, England, Dec. 1991.

[Audsley93] N.C. Audsley, K.W. Tindel, and A. Burns, “The end of the road for static cyclic scheduling”, Proceedings of 5th Euromicro Workshop on Real-Time Systems, pp. 36-41, Oulu, Finland, 1993.

[Bagrodia91] Rajive L. Bagrodia and C-C Shen, “MIDAS: Integrated design and simulation of distributed systems”, IEEE Transactions on Software Engineering, vol. 17, no. 10, Oct. 1991.

[Bannister83] J.A. Bannister and K.S. Trivedi, “Task allocation in fault-tolerant distributed systems”, in Acta Informica, vol. 20, pp. 261-281, 1983.

[Bause91] F. Bause and P. Buchholz, “Protocol analysis using a timed version of SDL”, Third Int. Conf. on Formal Description Techniques, Spain, 1991.

[Bause93] F. Bause and P. Buchholz, “Qualitative and quantitative analysis of timed SDL specifications”, Kommunikation in Verteilten Systemen, N. Gerner et al. (eds.), Springer-Verlag, pp. 486-500, 1993

[Bellcore95] Bellcore, SONET Bidirectional Line-Switched Ring (BLSR) equipment generic criteria, GR-1230-CORE, Issue 2, November 1995.

[Bettati94] R. Bettati, “End-to-end scheduling to meet deadlines in distributed systems”, Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, USA, March 1994.

[Biermann76] A.W. Biermann and R. Krishnaswamy, “Constructing programs from example computations”, IEEE Transactions on Software Engineering, vol. 2, pp. 141-153, 1976.

[Blanning95] R. Blanning, “A decision support framework for scenario management”, in Proc. 3rd Intl. Conf. on Decision Support Systems, 1995.

[Bokhari79] S.H. Bokhari. “Dual Processor Scheduling with Dynamic Reassignment”. IEEE Transactions on Software Engineering, vol. 5, pp. 341-349, July 1979.

[Bokhari81] S.H. Bokhari. “A shortest tree algorithm for optimal assignment across space and time in distributed processor system”. IEEE Transactions on Software Engineering, 7(11):583-589, Nov. 1981 .

[Bokhari87] S.H. Bokhari. “Assignment problems in Parallel and Distributed Computing”. Kluwer Academic Publishers, 1987.

[Braek93] R. Braek, O. Haugen, “Engineering Real Time Systems - An Object Oriented Methodology using SDL”. BCS Practitioner Series, Prentice Hall, 1994.

[Buchard94] A. Buchard, J. Liebeherr, Y. Oh, and S.H. Son, “Assigning real-time tasks to homogeneous multiprocessor systems”, Tec. Rep. CS-94-01, University of Virginia, Jan. 1994.

[Buchard95] A. Buchard, J. Liebeherr, Y. Oh, and S.H. Son, “New strategies for assigning real-time tasks to multiprocessor systems”, IEEE Transactions on Computers, vol. 44, pp. 1429-1442., 1995.

[Bui96] T. Bui, D. Kersten, and P.-C. Ma, “Supporting negotiation with scenario management”, in Proc. 29th Annual Hawaii Intl. Conference on System Sciences, vol. III, pp. 209-218, 1996.

[Buhr96] R.J.A Buhr and R.S. Casselman, Use Case Maps for object-oriented systems. Prentice Hall, 1996.

[Cameron97] Doanld Cameron, “Scenario design paradigm: synthesizing a system from scenarios”, Nortel Telecom internal report, March 1997.

[Carrol95] J. Carroll, “The scenario perspective on system development”, in Scenario-Based Design: Envisioning work and Technology in system Development (J. Carroll, ed.), John Wiley and Son, 1995.

[Chu87] Wesley W. Chu and L. M. Lan, “Task allocation and precedence relations for distributed real-time systems”, IEEE transactions on Computers, 36(6):667-679, June 1987.

[Coffman78] E.G. Coffman, M.R. Garey and D.S. Johnson, “An application of bin-packing to multiprocessor scheduling”, SIAM J. Comput., vol. 7, pp. 1-17, Feb. 1978.

[Coli95] M. Coli and P. Palazzari, “A new method for optimization of allocation and scheduling in real time applications”, Proc. Seventh Euromicro Workshop on Real Time Systems, pp. 262-269, June 1995.

[Davari85] S. Davari and S. Dhall, “On a real-time task allocation problem”, in Proc. of the 7th Hawaii International Conference on System Sciences, 1985.

[Davari86a] S. Davari and S. Dhall, “An on line algorithm for real-time task allocation”, in Proc. of the IEEE Real-Time System Symposium, pp. 194-200, 1986.

[Davari86b] S. Davari and S. Dhall, “On a periodic real-time task allocation problem”, in Proc. of the 19th Annual International Conference on System Sciences, pp. 133-141, 1986.

[Dhall77] S.K. Dhall, “Scheduling periodic-time-critical jobs on single processor and multiprocessor computing systems”, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1977.

[Dhall78] S.K. Dhall and C.L. Liu, “On a real-time scheduling problem”, in Operations Research, vol. 26(1), pp. 127-140, Feb. 1978

[Diefenbruch95] M. Diefenbruch, E. Heck, J. Hintelmann, and B. Muller-Clostermann, “Performance evaluation of SDL systems adjunct by queueing models”, Seventh SDL Forum, SDL’95, 1995.

[Diefenbruch98] M. Diefenbruch, and B. Muller-Clostermann, “Queueing SDL: A language for the functional and quantitative specification of distributed Systems”, Workshop on Performance and Time in SDL and MSC, Univ. Erlangen-Nurnberg, 1998, Workshop Proceedings.

[Elsayed98] H.M. El-Sayed, D. Cameron, and C.M. Woodside, “Automated performance modeling from scenarios and SDL designs of distributed systems”, In Proc. of the Int. Symposium on Software Engineering for Parallel and Distributed Systems (PDSE’98), Kyoto, April 1998.

[Etemadi96] Reza Etemadi, “End-to-end scheduling in hard real-time multiprocessor systems”, Ph.D. Thesis, Dept. of Systems and Computer Engineering, Carleton University, 1996.

[Franks95] G. Franks, A. Hubbard, S. Majumdar, D. Petriu, J. Rolia, and C.M. Woodside, “A toolset for performance engineering and software design of client-server systems”, Performance Evaluation, 24 (1-2):117-135, November 1995.

[Franks97] G. Franks, “Performance analysis of distributed server systems”, Ph.D. thesis proposal, Dept. of Systems and Comp. Eng., Carleton University, Feb. 1997.

[Garcia95] J.J.G. Garcia and M. Gonzalez Harbour, “Optimized priority assignment for task and messages in distributed hard real-time systems”, Proc. of the IEEE Workshop on Parallel and Distributed Real-Time Systems, California, pp. 124-132, April 1995.

[Garey79] M.R. Garey and D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, New York: W. H. Freeman and Company, 1979.

[Gomaa93] H. Gomaa, Software Design Methods for Concurrent and Real-Time Sytems, Addison-Wesley, 1993.

[Hsia94] P. Hsia, J. Samuel, J. Gao, Y. Toyoshima, and C. Chen, “Formal approach to scenario analysis”, IEEE Software, pp. 33-41, Mar. 1994.

[Hou92] C.J. Hou and K.G. Shin, “Allocation of periodic task modules with precedence and dead-line constraints in distributed real-time systems”, in Proc. of the Real-time system symposium, pp. 146-155, 1992

[Houstis90] C.E. Houstis, “Module allocation of real-time applications for distributed systems”, IEEE Transactions on Software Engineering, vol. 16, pp. 699-709, July 1990.

[Hrischuk95] C. Hrischuk, J. Rolia, C.M. Woodside, “Automated generation of software performance model using an object-oriented prototype”, International Workshop on Modeling and Simulation, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS ‘95), pp. 399-409, Durham, NC, 1995.

[Hrischuk99] C. Hrischuk, C.M. Woodside, J. Rolia and R. Iversen, “Trace-based load characterization for generaling software performance models”, IEEE Transactions on Software Engineering, vol. 25, no. 1, January 1999.

[Huang85] James P. Huang, “Modeling of software partition for distributed real-time applications”, IEEE Transactions on Software Engineering, vol.11, no. 10, October 1985.

[ISO87a] ISO DIS8807, “Lotos. A formal description technique”, 1987.

[ISO87b] ISO DIS9074, “Estelle: A formal description technique based on extended state transition model”, 1987.

[ITU93a] ITU-T: Z.100, “Specification and Description Language (SDL)” ITU, 1993.

[ITU93b] ITU-T: Z.100, “Message Sequence Chart (MSC)” ITU, 1993.

[Iyer89] V. R. Iyer and H. A. Sholl, “Software partitioning for distributed, sequential, pipelined applications”, IEEE Transactions on Software Engineering, vol. 15, no. 10, 1989.

[Jacobson92] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, Object Oriented Software Engineering-A Use Case Driven Approach, Addison Wesley, 1992.

[Jedrysiak94] S. Jedrysiak, G. Suwala, “Scenario-design system paradigm, a highly-automated approach”, Nortel Design Forum 1994.

[Jedrysiak96] S. Jedrysiak, D. Cameron, G. Suwala, “Rapid system design = executable requirements + automated reuse”, Nortel Design Forum 1996

[Jonsson97] Jan Jonsson and Kang G. Shin, “Deadline assignment in distributed hard real-time systems with relaxed locality constraints”, In Proc. of the IEEE Int. Conf. on Distributed Computing Systems, May 27-30, pp. 432-440, Baltimore, Maryland, 1997.

[Kao93] B. Kao and H. Garcia-Molina, “Deadline assignment in a distributed soft real-time system”, Proc. of the IEEE International Conference on Distributed Computing Systems, Pittsburgh, Pennsylvania, pp. 428-437, May 1993.

[Kao94] B. Kao and H. Garcia-Molina, “Subtask deadline assignment for complex distributed soft real-time systems”, Proc. of the IEEE International Conference on Distributed Computing Systems, Poznan, Poland, pp. 172-181, June 1994.

[Koskimies94] K. Koskimies and E. Makinen, “Automatic synthesis of state machines from trace diagrams”, Software-Practice and Experience, vol. 24(7), pp. 643-658, July 1994.

[Lam96] Kam-yiu Lam, Victor Lee, S. Hung and B.C. Kao, “Impact of priority assignment on optimistic concurrency control in distributed real-time databases”, pp. 128135, IEEE 1996.

[Lee92] C-H Lee D. Lee and M. Kim, “Optimal task assignment in line array networks”, IEEE Transactions on Computers, vol. 41, no. 7, pp. 877-888, July 1992.

[Lee97] C-H Lee and Kang G. Shin, “Optimal task assignment in homogeneous networks”, IEEE Transactions on Parallel and Distributed Systems, vol. 8, no. 2, Feb. 1997.

[Lehoczky89] J. Lehoczky, L. Sha, and Y. Ding, “The rate monotonic scheduling algorithms: exact characterization and average case behavior”, In Proc. of the 10th IEEE Real-Time Systems Symposium, pp. 166-171, 1989.

[Lehoczky90] J.P. Lehoczky, “Fixed priority scheduling of periodic task sets with arbitrary deadlines”, Proceedings IEEE Real-Time Systems Symposium, pp. 201-209, 1990.

[Leung82] J.Y.T. Leung, J. Whitehead, “On the complexity of fixed-priority scheduling of periodic real-time tasks”, Performance Evaluation, 2, (4), pp. 237-250, Dec. 1982.

[Lin91] Feng-Tse Lin and Ching-Chi Hsu, “Task assignment problems in distributed computing systems by simulated annealing”, Journal of the Chinese Institute of Engineers, 14(5):537-550, Sept. 1991.

[Liu73] C.L. Liu and J.W. Layland, “Scheduling algorithms for multiprogramming in a hard real-time environment”, in Journal of the Assoc. Computing Mach., vol. 20(1), pp. 46-61, 1973.

[Lo88] V.M. Lo, “Heuristic algorithms for task assignment in distributed systems”, IEEE Transactions on Computers, vol. 20, pp. 1384-1397, Nov. 1988

[Ma82] P.R. Ma, E.Y. Lee, and M. Tsuchiya, “A task allocation model for distributed computing systems”, IEEE Transactions on Computers, 31:41-47, Jan. 1982.

[Martins95] J. Martins and J.-P. Hubaux, “A new methodology for performance evaluation based on the formal technique SDL”, Workshop on Design and Analysis of Real-Time Systems, Sponsored by Spacebel Informatique, pp. 73-88, Belgium, 1995.

[Mazzeo97] A.Mazzeo, N. Nazzocca, S. Russo and V. Vittorini, A systematic approach to the Petri Net based specification of concurrent systems”, Real-Time Systems, Vol. 13, pp. 219-236, 1997.

[Menace98] D. Menasce and H. Gomaa, “On a language based method for software performance engineering of client/server systems”. Proceedings of the First International Workshop on Software and Performance WOSP 98, Santa Fe, New Mexico, October 12-16, 1998.

[Menace99] D. Menasce and H. Gomaa, “A method for design and performance modeling of client/server systems”. to appear in the IEEE Transactions on Software Engineering.

[Mok84] A.K. Mok, “The decomposition of real-time system requirements into process models”, Proceedings of the IEEE 1984 Real-Time System Symposium, Texas, Dec. 1984, pp. 125-134

[Mok86] A.K. Mok, “The Von Neuman Straitjacket - the process construct”, ACM SIGSOFT Software Engineering Notes, Vol 11 No 4, Aug. 1986.

[Mutka95] M.W. Mutka and J.P. Li, “A tool for allocating periodic real-time tasks to a set of processors”, Journal of Systems and Software, vol. 29, pp. 135-148, 1995

[ObjecTime97] ObjecTime Limited, Kanata, Ontario, Canada. ObjectTime User's Manual, 1997.

[Oh95] Y. Oh and S.H. Son, "Allocating fixed-priority periodic tasks on multiprocessor systems", Real-Time Systems, vol. 9, pp. 207-239, 1995.

[Oslen94] A. Oslen, O. Faergemand, B. Moller-Pedersen, R. Reed and J.R.W. Smith, "Systems engineering using SDL-92", North Holland, 1994.

[Parnas67] D. Parnas and J.A. Darringer, "SODAS and a methodology for system design", in AFIPS Conf. Proc., Fall Joint Comput. Conf., Washington, DC: Thompson, vol. 31, pp. 449-474, 1967.

[Peng87] D.T. Peng and K.G. Shin, "Assignment and scheduling of communicating periodic tasks in distributed processing systems", Proceedings of the IEEE, vol. 75, pp. 547-562, May 1987.

[Peng89] D.T. Peng and K.G. Shin, "Static allocation of periodic tasks with precedence constraints in distributed real-time systems", In Proc. of the 9th Intl. Conf. on Distributed computing systems, pp. 190-198, 1989.

[Potts94] C. Potts, K. Takahashi, and A. Anton, "inquiry-based requirement analysis", IEEE Software, vol. 11, pp. 21-32, Mar. 1994.

[Purimetla94] B. Purimetla, R.M. Sivasankaran, J.A. Stankovic, K. Ramamritham and D. Towsley, "Priority assignment in real-time active databases", pp. 176-184, IEEE 1994.

[Ram90] K. Ramamritham, "Allocation and scheduling of complex periodic tasks", In Proceedings of the 10th International conference on distributed Computing Systems, pages 108-115, Paris, France, 1990.

[Ram95] K. Ramamritham, "Allocation and scheduling of precedence-related periodic tasks", IEEE Transactions on Parallel and Distributed Systems, 6(4): 412-420, April 1995.

[Regnell95] B. Regnell, K. Kimbler, and A. Wesslen, "Improving the Use Case driven approach to requirements engineering", In Proceedings of Second Int. Symposium on Requirements Engineering, York, U.K., IEEE Computer Society Press, pp. 40-47, March 1995.

[Regnell96] B. Regnell, M. Andersson, and J. Bergstrand, "A hierarchical use case model with graphical representation", In Proceedings of ECBS'96, IEEE Second International Symposium of Computer-Based Systems, 1996.

[Riddle72] W.E. Riddle, The Modeling and Analysis of Supervisory Systems, Ph.D. thesis, Stanford University, Mar. 1972.

[Rolia95] J. R. Rolia and Kenneth Sevcik, “The method of layers”, IEEE Transactions on Software Engineering, Vol. 21, No. 8, pp. 689-700, 1995.

[Roman87] G.C. Roman, “Specifying software/hardware interactions in distributed systems”, In Proc. Int. Conf. Software Eng., pp. 126-139, May 1987.

[Rolland98] C. Rolland et al., “A proposal for scenario classification framework”, Requirements Engineering Journal, Vol. 3, No. 1, Springer Verlag, 1998.

[Rumbaugh91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, Object Oriented Modeling and Design, Prentice-Hall, 1991.

[Sager89] G. Sager, Anil K. Sarje, and Kamal U. Ahmed “Task allocation techniques for distributed computing systems: a review”, Journal of Microcomputer Applications. 12(2):97-105, April 1989

[Santos97] J. Santos, E. Ferro, J. Orozco, and R. Cayssials, “A heuristic approach to the multitask-multiprocessor assignment problem using the empty-slots method and rate monotonic scheduling”. Real-Time Systems, 13, 167-199 (1997).

[Saksena96] M. Saksena and S. Hong, “An engineering approach to decomposing end-to-end delays on a distributed real-time system”, Proc. of the IEEE Workshop on Parallel and Distributed Real-Time Systems, Hawaii, pp. 244-251, April 1996.

[Saracco89] R. Saracco, J.R.W. Smith and R. Reed, Telecommunications Systems Engineering using SDL, Elsevier Science Publishers B.V., 1989.

[Sarje91] A.K. Sarje and G. Sagar, “Heuristic models for task allocation in distributed computer systems”. IEE Proceedings, Part E, [Computer and Digital Techniques], 138(5):313-318, Sept. 1991.

[Scratchley99] W.C. Scratchley and C.M. Woodside, “Evaluating concurrency options in software specifications”, Submitted to MASCOTS 99.

[Shen85] Chien-Chung Shen and Wen-Hsiang Tsai, “A graph matching approach to optimal task assignment in distributed computing systems using a minimax criteria”, IEEE Transactions on Software Engineering, 34(3):197-203, March 1985.

[Sherman95] W.Sherman, D.J. Paulish, M. Klinger and W. Liao, “Scenario-driven software design”, Electronic Design, October 24, 1995.

[Smith90] C.U. Smith, Performance Engineering of Software Systems, Addison-Wesley Publishing Co., New York, NY, 1990.

[Smith97] C.U. Smith and Lloyd Williams, “Performance engineering evaluation of object-oriented systems with SPE-ED”, Computer Performance Evaluation:

Modeling Techniques and Tools, No. 1245, (R. Marie, et. al. eds.), Springer-Verlag, Berlin, 1997.

[Sngauinetti79] J. Snaguinetti, “A technique for integrating simulation and system design”, in Conf. on Simulation, Measurement and Modeling of Comput. Sys., Aug. 1979.

[Stone77] Harold S. Stone, “Multiprocessor scheduling with the aid of network flow algorithm”, IEEE Transactions on Software Engineering, 3(1):85-93, Jan. 1979

[Stone78a] Harold S. Stone, “Critical load factors in distributed computer systems”, IEEE Transactions on Software Engineering, vol. 4, pp. 254-258, May 1978.

[Stone78b] Harold S. Stone and S.H. Bokhari, “Control of distributed processes”, Computer, vol. 11, pp. 97-106, July1978.

[Storch93] M.F. Storch and J.W.S. Liu, “Heuristic algorithms for periodic job assignment”, In Proceedings of the Workshop on Parallel and Distributed Real-time Systems, pp. 245-251, Apr. 1993.

[Strosnider96] Jay K. Strosnider, “Performance Engineering Real-Time/Multimedia Systems”, “http://www.ece.cmu.edu/afs/ece/usr/jks/web/home.html”

[Sun96] Jun Sun, Fixed Periodic Scheduling of Periodic Tasks with End-To-End Deadlines, Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, USA, March 1996.

[Tantawi85] A.N. Tantawi and D. Towsley, “Optimal static load balancing in distributed computer systems”, Journal of the ACM, vol. 32, pp. 445-465, April 1985.

[Telelogic96] SDT 3.1 User’s Guide, SDT 3.1 Reference Manual. Telelogic Malom AB, Sweden, 1996

[Thiel96a] A. Mitschele-Thiel, “Methodology and tools for the development of high performance parallel systems with SDL/MSC”, Proc. Software Engineering for Parallel and Distributed Systems, 1996

[Thiel96b] A. Mitschele-Thiel, P. Langendofer, and R. Henke, “Design and optimization of high-performance protocols with the DO-IT toolbox”, Proc. of FORTE/PSTV ‘96.

[Thiel99] A. Mitschele-Thiel, and B. Muller-Clostermann, “Performance engineering of SDL/MSC systems”, 9th SDL-Forum, Montreal, June 1999.

[Tindel92] K.W. Tindel, A. Burns, and A.J. Wellings, “Allocating hard real-time tasks: an NP-hard problem made easy”, Real-Time Systems, 4(2):145-165, June 1992.

[Towsley86] Don Towsley, “Allocating programs containing branches and loops within a multiple processor system”, IEEE Transactions on Software Engineering, 12:1018-1024, Oct. 1986

[UML99] UML Revision Task Force: OMG Unified Modeling Language Specification, version 1.3 beta R1, June1999. http://uml.systemhouse.mci.com/

[Verilog96] Verilog. ObjectGEODE - Technical Documentaion, 1996.

[Weiden98] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer, “Scenario usage in system development: a report on current practice”, IEEE Software, pp. 34-45, March 1998.

[Wohlin91] Claes Wohlin, “Performance analysis of SDL systems from SDL specifications”, SDL design forum 1991.

[Woodside89] C.M. Woodside, “Throughput calculation for basic stochastic rendezvous networks”, Performance Evaluation, Vol. 9, No. 2, pp. 143-160, 1989.

[Woodside93] C.M. Woodside and G.M. Monforton, “Fast allocation of processes in distributed and parallel systems”, IEEE Transactions on Parallel and Distributed Systems, vol. 4, no. 2, Feb. 1993.

[Woodside95a] C. M. Woodside, J. E. Neilson, D. C. Petriu, and S. Majumdar, “The stochastic rendezvous network model for performance of synchronous client-server-like distributed software”, IEEE Transactions on Computers, 44(1):20-34, January 1995.

[Woodside95b] C. M. Woodside, “Three-view model for performance engineering of concurrent software”, IEEE Transactions on Software Engineering, vol. 21, no. 9, Sept. 1995.

[Woodside98] C. M. Woodside, C.E. Hrischuk, B. Selic, S. Bayarov, “A wideband approach to software performance prediction and improvement”, 1st International Workshop on Software and Performance (WOSP’98), October 10-12, 1998.

[Xin88] Huang Xin, Zhang Hong, and X. Cai, “Heuristic software partitioning algorithms for distributed real-time applications”, IEEE 1988.

[Zurcher68] F. Zurcher and B. Randell, “Iterative, multi-level modeling - a methodology for computer system design”, in Proc. IFIP Congress ‘68, pp. 867-871, 1968.

THIS IS THE LAST PAGE MARKER