COMPOSITIONAL FRAMEWORK FOR THE GUIDED EVOLUTION OF TIME-CRITICAL SYSTEMS

SHIKUN ZHOU

A thesis submitted in partial fulfilment of the requirements of De Montfort University for the degree of Doctor of Philosophy

Software Technology Research Laboratory

De Montfort University

May 2002

Abstract

Most of the time-critical computer systems are special-purpose and complex, and are typically embedded in larger systems, such as avionics and robotics control systems. The engineering of time-critical systems poses significant challenges to their 'correct' specification, design, development and evolution. Because of the complexity of time-critical systems, the likehood of subtle errors is much greater than other computer systems and some of these errors could have catastrophic consequences such as loss of life, money, time or damage to the environment. It has been recognised that the use of formal methods, in the life-cycle of time-critical systems, is fundamental.

The thesis proposes an approach, based on a formal method (known as Interval Temporal Logic (ITL)), for engineering time-critical systems, rapidly, efficiently and above all, correctly. The approach uses an integrated framework to deal with the life-cycle of time-critical systems. The proposed framework integrates conventional approaches and formal technologies for engineering time-critical systems.

Based on this framework, the thesis then focuses on using ITL and its executable subset, Tempura, for the development and evolution of time-critical systems development and evolution. An important issue during this evolution is to manage 'change', as well as to cope with its propagation. This is called *guided evolution* in the thesis. The proposed approach aims to validate and analyse time-critical system's behaviours of interest, such safety, liveness, as well as analyse timing behaviours and ensure the correctness of the timing properties. The validation and analysis are performed at run-time. The assumption/commitment paradigm has been adopted in the thesis. The assumption/commitment technique is valuable as a compositional principle to be used during time-critical systems development and evolution. Behavioural properties expressed in ITL can be validated and tested compositionally. The framework presented in the thesis is language independent.

The proposed approach can deal with both sequential and parallel time critical systems. This is a particular challenging research area because within such a system the functional behaviour and non-functional timing requirements are combined, implicit and can be difficult to validate and analyse.

A prototype tool is developed for three purposes: to test the approach, to speed and to scale up time-critical systems development and evolution based on the proposed approach. Two case studies, including a post office letter sorting system and an assembly line control system, are used for experiments with the approach and the prototype tool.

Conclusion is drawn based on analysis of experiments, which shows that the proposed approach is feasible and promising in its domain. Further research directions are also discussed.

Contents

List of Figures

List of Tables

Acknowledgements

I wish to express my most profound thanks to my supervisors Professor Hussein Zedan and Dr. Antonio Cau for their invaluable advice, support and encouragement during my three year study. Without any of these, the work in this thesis would only be impossible. There are so many intensive discussions that impressed me so much.

Meanwhile, I would like to thank colleagues at Software Technology Research Laboratory and Department of Computer Science in De Montfort University for their support and feedback, and for providing such a stimulating and friendly working atmosphere. There are too many to list individually. The regular seminars provide us with a good opportunity to communicate, discuss and co-stimulate.

I would also like to thank the Research Office in De Montfort University for their outstanding management.

Finally, I must thank my parents for all their memorable support and encouragement, which are too precious to forget.

Declaration

I declare that the work described within this thesis was originally taken by me between the dates of registration for the degree of Doctor of Philosophy at De Montfort University, November 1998 to May 2002.

Publications

H. Zedan, S. Zhou, N. Sampat, X. Chen, A. Cau, and H. Yang, *K-Mediator: Towards Evolving Information Systems*, in IEEE Proceedings of International Conference on Software Maintenance'2001 (ICSM2001), 2001.

H. Zedan, A. Cau, and S. Zhou, *A Calculus for Evolution*, in Proceedings of The Fifth International Conference on Computer Science and Informatics (CS&I'2000), 2000.

S. Zhou, H. Zedan, and A. Cau, *A Framework For Analysing The Effect of 'Change' In Legacy Code*, in IEEE Proceedings of International Conference on Software Maintenance'99 (ICSM99), 1999.

S. Zhou and H. Yang, *An Approach to Measuring Reverse Engineering*, in Proceedings of the Fifth International Conference for Young Computer Scientists (ICYCS99), August, 1999.

S. Zhou, H. Yang, P. Luker, W.C. Chu and X.He, *A Useful Approach to Developing Reverse Engineering Metrics*, in Proceedings of the 23rd IEEE Computer Software and Application Conference (COMPSAC99), 1999.

Chapter 1

Introduction

1.1 Purpose of Research and Overview of the Problem

The engineering of time-critical systems poses significant challenges to their *'correct'* specification, design and development. In such systems, the time, at which each input is processed or output is produced, is critical. In general, time-critical systems are characterised by the fact that severe consequences will result if functional as well as timing properties of the system are not satisfied. Most of the time-critical computer systems are special-purpose and complex, require a high degree of fault tolerance, and are typically embedded in a larger system [118]. Avionics, robotics and process control are all examples of time-critical computing [118].

An important aspect in the development of a time-critical system is how to cope with its "evolution". The evolution of a time-critical system happens not only after the first delivery of the system but also in the early stage of its life-cycle, i.e. evolutionary development. The evolution of a time-critical system could be due to changes in the original requirements, adopting a different hardware platform or to improve its performance in its entire life-cycle. Rapid development causes continuous changes in the software life-cycle. Because of the timing aspects, changes in a time-critical system cause even more troubles to developers of time-critical systems than in other applications. Because of their (time-critical systems) complexity, the likelihood of subtle errors is much greater and some of these errors could have catastrophic consequences such as loss of life, money, or damage to the environment. Hence, how to respond to '*change*' is a fundamental issue in the development of time-critical system . This response must be undertaken rapidly, efficiently and, above all, correctly. Responding to changes requires understanding the functionality, as well as timing aspects, of the system, identification of the necessary changes and then apply the changes.

Another important issue in managing change is to establish mechanisms to cope with its propagation. The change is often made to a specific part of the system. After the change is made, that part may no longer be compatible with other parts of the system, as it may no longer provide what was originally expected or it may now require different services for the rest of the system. These dependencies need to be checked, validated and re-established if they are lost. The process in which the change spreads through the software is often called the *ripple effect* of change [136]. Various techniques have been proposed to model change [6, 17] and its impact [99, 107, 106] for non-real time systems. The prediction of the size and location of change has also been considered (e.g. [40]).

In this research, we concentrate on following aspects of change management, especially, managing impact of change, mainly, identifying, specifying and controlling ripple effects of changes:

- *Change analysis*: to analyse and specify the change correctly using formal methods;
- *Impact analysis*: to analyse impact of change via analysing, revealing and specifying the relationships between different system components and changes by using compositional theory and assumption/commitment framework, furthermore, to control ripple effects;
- *Ripple effect control*: The use of compositional theory and assumption/commitment framework can reduce the ripple effect (we will define this in next section), because isolating a target sub-system from its surrounding sub-systems during engineering processes becomes possible and effective, which will decrease the level of likely ripple effects causing by neighbouring sub-systems or draw a clear picture to ripple effects.

In this thesis, we are going to discuss how to tackle the development and evolution of time-critical systems rapidly, efficiently and, correctly, based on the following observations:

- Due to the very complex nature of time-critical systems, their development and evolution processes must be repeatable, well-defined, managed, and potentially optimised.

- It has been recognised that the use of formal methods, in the development of such systems, is fundamental if "correctness" is to be assured. Using formal approaches increases our understanding of a system by revealing inconsistencies, ambiguities, and incompleteness that might otherwise go undetected [77, 135]. Hence, formal techniques are crucial for the engineering of *correct* time-critical systems.

- In a time-critical system, the functional behaviours of the system and the non-functional timing requirements are highly coupled, implicitly and can be very difficult to analyse and validate. Attempting to analyse and validate such a system is a particular challenging research area.

- Due to the unmanageable size and extreme complexity of time-critical systems, an ideal solution is to partition the whole system into smaller and manageable sub-programs or components. These sub-programs or components will be treated separately, based on an assumption about the behaviour of its surrounding sub-programs, components or even the environment of the whole systems. Therefore, compositional theories shall be adopted and developed.

- In the development and evolution process of a time-critical system, most timing requirements will not be proved to be satisfied until full implementation of the system, i.e., testing of the source code in real working conditions. Analysis and validation must be performed at run time before the final deployment of the system.

The terms used in the chapter, such as time-critical systems, behaviours, compositional theories, assumption/commitment framework, etc., will be defined in the following chapters.

1.2 Scope of the Thesis and Original Contribution

This thesis aims to present a sound technique, together with its supporting tool, for engineering time-critical systems rapidly, efficiently and, above all, correctly. The thesis concentrates on engaging formal techniques to handle development and evolution of time-critical systems. The scope of research includes:

- The architectural design of the integrated approach: a phase-based methodology has been identified in the approach, supporting guided evolution, i.e., evolving time-critical systems under a controlled manner.

- The formalisation of the notion of *run-time analysis*: run-time analysis for time-critical systems will be formally defined. Specification-based assertion points technique has been developed for run-time analysis. The proposed run-time analysis technique has been implemented in a prototype tool.

- The development of compositional theory: the role of compositionality has been identified and rules to conduct composition of time-critical systems are developed aiming at evolutionary development of time-critical systems.

- Implementation of a prototype tool and experimentation with case studies: a system is developed to illustrate and support the proposed approach. Another purpose of the prototype tool is to implement the developed theories and rules for guided evolution. A number of case studies are used for experiments with the approach and the prototype system.

The original contribution of the thesis lies in two aspects:

- Guided evolution of time-critical systems. The contents of guided evolution have been identified and defined in the thesis. During the evolutionary development of time-critical system, guided evolution provides a technical basis for a repeatable, well-defined, managed, and potentially optimised development process. It first addresses a general architectural design of an integrated engineering framework for handling evolutionary development of time-critical systems. This involves crossing levels of time-critical systems, from specification in mathematical manners to source code in different programming languages (language independent). Time-critical systems' behaviours of interest can be analysed and validated in any stage of evolutionary development. The validation and analysis are performed within a *single* logical framework. As the main guideline provided in the evolution of the time-critical systems, a set of extendible compositional rules offer a repeatable and well-manageable way to handle evolutionary development of time-critical systems.

- Run-time analysis. It is a key part in this thesis. One mechanism, namely, assertion points, are used to generate run-time data (assertion data), which fully reflects run-time behaviours of the time-critical system. An analysing and validating mechanism has been designed to capture and analyse assertion data. The analysing and validating mechanism then analyses and validates run-time behaviours with respect to formal specification of the system. Errors will be reported during the system run. The run-time analysis does not only report an error but also indicate the location of the error.

1.3 Criteria for Success

The following criteria are given to judge the success of the research described in this thesis:

- For a "living" time-critical system, what is the most specific characteristic, distinguishing it from other system from the perspective of evolution and making a higher potential for producing impact of change than other conventional systems?
- Can we have a systematic way to cope with the specific characteristic of time-critical systems and its evolutionary life-cycle?
- Can timing or functional behaviours of interest of a time-critical system be captured, analysed and validated efficiently and correctly under its real working environment in real-time?
- Can a time-critical system be developed under a repeatable base?
- How easy is it to manage evolutionary development of a time-critical system using the proposed approach?

- Is the approach feasible for realisation? For example, is it possible to build a tool based on the approach?

1.4 Thesis Structure

The thesis is organised as follows:

- Chapter 1 gives the background, motivation, scope and original contribution of the thesis.
- Chapter 2 provides an overview of the current state of the art in the development and evolution of time-critical systems, related formal notation and methods, scheduling, software evolution, and in particular, their intersection, time-critical systems' engineering.
- Chapter 3 discusses the related work, especially those involving usage of formal techniques. Reasons of choosing Interval Temporal Logic (ITL) and its workbench as formal base of the proposed approach in the thesis have been given in the chapter. New extended work of ITL workbench with respect to time-critical systems engineering has also been described in this chapter.
- Chapter 4 explores the proposed approach in detail, including definition and contents of guided evolution, a step-by-step methodology, assertion points technique and visualisation of time-critical systems.

- Chapter 5 describes run-time analysis, compositional guidelines and timing analysis guidelines of the guided evolution.
- Chapter 6 is about realisation of the proposed approach by building a tool, namely, AnaTempura. The chapter covers the tool's general system architecture, key tool components and user interface.
- Chapter 7 deals with case studies, which include three case studies with different aspects, namely, an experiment of the general fulfilment of the approach with using AnaTempura, a test of application of compositional guidelines in a time-critical system with shared variable-based parallelism and an implementation of the approach towards another time-critical system with message-based parallelism.
- Chapter 9 discusses the proposed approach and the supporting tool according to a set of criteria. Conclusion is drawn based on this discussion, and prospective further work is also discussed.

Chapter 2

Time-critical Systems: Development and Evolution

Objectives:

To give an overview to time-critical systems

To discuss key issues in time-critical systems development

To investigate relevant formal methods

To conclude main problems of time-critical systems evolution

2.1 Introduction

Any computing system, either a hardware or a software system, will inevitably grow in scale and functionality. Time-critical systems tend to be *large* and *complex, functioning in distributed and dynamic environments*, and have complex timing constraints. These systems have brought significant challenges to a wide range of software engineering disciplines.

Real-time systems differ from traditional systems in that deadlines or other explicit timing constraints are attached to tasks, the systems are in a position to make compromises and faults, including timing faults, that may have catastrophic consequences [118].

Any system, by its nature, tends to follow an evolutionary development and any time-critical system is no exception. This evolutionary development is regarded as being divided into *corrective* actions to fix latent defects, *adaptive* actions to deal with changing environments, and *perfective* actions to accommodate new requirements [125]. A major goal in evolution is to enable the system to be operated correctly, despite its complexity, throughout the software life-cycle. One way of achieving this goal is by using *formal methods*, which are mathematically-based languages, techniques and tools for specifying and verifying both hardware and software systems. They have the clear advantage that precise descriptions can be made. Moreover, it is possible to prove that certain necessary properties hold. Most of the existing work which uses formal methods concentrates on the high abstract level, i.e. specifications, without regard to the source code. Also, the much stricter timing requirements for time-critical systems impose more demands on the implementation [73]. Most timing requirements will not be proved to be satisfied until full implementation of the system. Timing analysis must be employed at any time from the phase of defining the requirements and specifying the system to the final deployment of the system.

This chapter investigates the current situation of software evolution, real-time systems and formal methods. It proposes the basic criteria for formal methods to be applied in the time critical systems domain. Among the range of the application areas of time critical systems, this thesis concentrates on the evolutionary development of time critical systems. Based on the proposed criteria, the existing popular formal methods, especially temporal logic, are investigated and assessed.

2.2 Time-critical Systems

In the real world, more and more vital applications such as nuclear power stations, flight control software for airplanes, the space shuttle avionics systems, the space station, etc., are of a time-critical nature. Time-critical systems are characterised by quantitative timing properties relating occurrences of events [73]. In general, time-critical systems are part of real-time systems, but meeting the deadline of tasks in a time-critical system is considered as critical as these tasks. Terms such as "hard" and "soft" real-time systems are sometimes used. In this section, we will describe what a real-time system is; provide an overview; discuss some emerging principles and primitives for real-time systems and time-critical systems.

2.2.1 What is a Real-time System?

Real-time systems are characterised by the fact that severe consequences will result if functional as well as timing properties of the system are not satisfied. Real-time systems are increasingly being used to ensure the effective operation of a wide range of human activities, including *administration, financial management, manufacturing and process control* [98]. The *Oxford Dictionary of Computing* defines a real-time system as:

> Any system in which the time at which output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness.

Some of the characteristics of real-time systems are [23]:

- *Large and complex.* Although real-time software is often complex, features, such as information hiding, separate compilation and abstract data types provided by real-time languages and environments enable these complex systems to be broken down into smaller components which can be managed effectively.
- *Manipulation of real numbers.* A fundamental requirement of a real-time programming language, therefore, is the ability to manipulate real numbers, i.e. many real-time applications (for example, signal processing, simulation and process control) require numerical computation facilities beyond those provided by integer arithmetic.
- *Extremely reliable and safe.* The complexity of real-time systems exacerbates the reliability problems; not only must expected difficulties inherent in the application be taken into account but also those introduced by faulty software design.
- *Concurrent control of separate system components.* A major problem associated with the production of software for systems which exhibit concurrency is how to express that concurrency within the structure of the program.

- *Real-time control facilities.* Response time is crucial in any real-time system. It is very difficult to design and implement systems which will guarantee that the appropriate output will be generated at the appropriate times under all possible conditions.

- *Interaction with both hardware and software interfaces.* The nature of real-time systems requires that computer components interact with the external world. For example, they need to monitor sensors and control actuators for a wide variety of real-world devices.

2.2.2 Time-critical Systems

Before proceeding further, it is worth spending some time clarifying the term "time-critical system". It is well-known that real-time systems all have in common the notion of timeliness. Depending on the criticality level of timeliness, real-time systems are often distinguished between *hard* and *soft* real-time systems. Burns [23] defines that:

> **hard real-time systems** are those where it is absolutely imperative that responses occur within the specified deadline, whilst
>
> **soft real-time systems** are those where response times are important but the system will still function correctly if deadlines are occasionally missed.

For example, a flight control system of a combat aircraft is a hard real-time system because a missed deadline could lead to a catastrophe, whereas a data acquisition system for a process control application is soft as it may be defined to sample an input sensor at regular intervals but to tolerate intermittent delays [23].

Hard real-time tasks are also sometimes called *time-critical tasks*. Systems with such tasks are often embedded in a large target system (consisting of both software and hardware) of which the tasks are an inseparable part. Such a system must cooperate with its surrounding, real-time, real-world environment. Computers are being used increasingly in safety-critical systems because of the added flexibility and decreased costs that they can generate. However, many accidents have been blamed on the use of computers and especially on the software in them.

Critical systems are defined as [63]:

> A computer, electronic or electro-mechanical system whose failure may cause injury or death to human beings, e.g., an aircraft or nuclear power station control system.

Safety is closely coupled to the notion of risk. Charette [28] defines risk as an event or action:

- Having a loss associated with it.
- Where uncertainty or chance is involved.
- Some choice is also involved.

Safety can then be defined as the freedom from exposure to danger, or the exemption from hurt, injury or loss. Criticalness of time-critical systems ties tightly with the timeliness. The safety of time-critical systems is coupled to the risks triggered by timing properties.

It is necessary to define a *time-critical system* more precisely depending on the relationships between such a system and its environment.

> A *time-critical system* must keep abreast with its environment, which is a *time-critical environment*, by reacting properly and timely to events occurring in the environment from the operation of the system. In such a system, the damage incurred by a missed deadline is greater than any possible value that can be obtained by correct and timely computation and may cause catastrophic results.

The term “ environment” has been defined as:

> The *environment* of a time-critical system embraces all external factors or forces, which include surrounding things, conditions or influences, especially affecting the existence or development of the system.

For example, the environment of a robot control system includes all sensors and actuators of the robot.

2.2.3 Timing issues of Time-critical Systems

Above all, it is necessary to investigate timing issues in time-critical systems. First of all, we need to clarify the notion of *timing properties*. Motus classifies timing properties into three groups [98]:

- **Performance-bound properties** which comprise integral time characteristics for the system as a whole, or for a part of it. Examples of this group are response time, time-out, and execution time for sequences or loops in programs.

- **The time-wise correctness of events and data** is concerned with execution time of programs and delays between events.

- **Time correctness of interprocess communication** is concerned with reactions in real-time systems, tending to be responsive to stimuli.

Timing behaviours of a system are usually realised by constructing a schedule for co-ordinating the execution of processes. A number of timing parameters are used to determine the behaviour of a system. These parameters will be used for scheduling and to express timing properties. They are [23]:

- *Start time:* the time instant when a task is activated.

- *Computation (or execution) time:* the time interval between the start time and the termination time of a task.

- *Deadline:* the upper limit for the termination of a task.

- *Activation period:* the interval between two successive start times for a task.

- Any *communication delays* incurred per message transferred.

- *average times* spent by tasks in queues.

2.3 Research Issues in Time-critical Systems

Due to the very complex nature of time-critical systems and the demands of their evolutionary development, the approaches and tools of evolutionary development of time-critical systems are still far from mature. In this section, research issues of evolutionary development of time-critical systems are discussed.

Formal Development. The high reliability requirements in time-critical systems have caused a movement away from informal approaches to the structured and, increasingly, the formal [23]. McDermid names three techniques [83]: *informal, structured* and *formal.* Informal methods usually make use of natural language and various forms of imprecise diagrams. When describing software in a natural language, three main problems, *ambiguity, incompleteness*, and *contradiction*, can occur. Structured methods often use a graphical representation, but unlike the informal diagrams these graphs are well defined. The graphical form may also have a syntactical representation in some well-defined language [23]. Although structured methods can be made quite rigorous, they cannot, in themselves, be analysed or manipulated. It is necessary for the notation to have a mathematical basis if such operations are to be carried out. Methods that have such mathematical properties are usually know as **formal**. They have the following clear advantage:

- Precise descriptions can be made in formal notations.
- They allow systems to be defined in abstract terms.

- They demand attention to issues of completeness and consistency, therefore reducing the chances of overlooking certain areas or situations which could cause errors or bugs [49].
- It is possible to prove that necessary properties hold.
- They allow the progressive refinement of an abstract specification into a concrete specification using well-defined rules. This leads to the possibility of generating programs from formal specifications automatically.
- Using formal descriptions it is possible to detect the deviations (intentional or otherwise) of a program from its original specification. It may be possible to create tools to carry out much of this detection work [49].

In this thesis, formal methods provide a solid theoretical foundation for integrating techniques in handling the evolutionary development of time-critical systems and building a practical formal tool.

Scheduling. A common characteristic of many real-time systems is that their requirements specification includes timing information in the form of deadlines. In a time-critical system the situation may indeed be catastrophic with actual damage resulting from an early or missed deadline. In general, there are two views as to how a system can be guaranteed to meet its deadline. One is to develop and use an extended model of correctness; the other focuses on the issue of scheduling [71]. The use of appropriate scheduling algorithms has been isolated as one of the semantic models used to describe the properties of real-time systems [118].

However, in this thesis, we are not going to describe how to develop scheduling theories. Instead, we give a review of timing diagrams technique, which is embodied in our own methodology of handling the evolution of time-critical systems.

Timing analysis. This research issue focuses on analysing timing properties via formal techniques and ensuring the correctness of timing properties of time-critical systems.

Timing analysis can be illustrated by looking at a typical list of questions which can be asked, in any given practical situation, about the characteristics of time [115]:

- Is time discrete or continuous?
- Is time unbounded?
- If time is continuous, is it dense, and if so, is it complete - in other words, can continuous time be modelled by rational or real numbers?
- Is time branching or linear; cyclic or acyclic?
- If time branches, should past and future be handled differently?

Run-time detection and verification technology. System design focuses on closing the "semantic gap" between the given application requirements and the chosen run-time hardware/software architecture. Time-critical system explicitly requires run-time mechanisms that guarantee upper bounds for the maximum execution time of critical tasks and for the maximum duration of the interprocess communication protocols [72]. Most of the timing properties of time-critical systems cannot be verified to be correct or not before full implementation of the entire system. There are a few general purpose methodologies providing solutions to this issue.

Change management of time-critical systems. Understanding the impacts of software change has been a challenge since software systems were first developed [17]. Formal methods could be helpful when handling change in time-critical software development and make change management more efficient. This technique is called *guided evolution.*

Compositional evolution. There are two reasons for handling evolutionary development of time-critical systems *compositionally.*

> Because of the size and complexity of time-critical systems, it is impossible to handle the whole system at once. The solution is to decompose the whole system into smaller, manageable units/components that are easier to handle and can be treated fairly *independently* of one another.
>
> Time-critical systems are characterised by a close interaction between the system and its environment. Modern time-critical systems even allow highly complex computer-environment interactions that humans are no longer able to control. Formal compositional techniques are extremely helpful to reveal interactions between the system and its environment and then to specify and verify these systems.

Program Visualisation of time-critical systems. Program visualisation comprises techniques where the program is specified in a conventional, textual manner, and pictorial representations are used to illustrate different aspects of the program, for instance, its run time behaviour [127]. Program visualisation is already a common technique, however, this technology, such as one of the first program animation systems, BALSA [20], and its descendants, is only applicable to sequential programs. New methods and tools have to be developed that are able to visualise the special kinds of information needed in the context of time-critical computing, for example, parallel behaviours.

2.4 Formal Notation and Methods

The debate about the use and relevance of formal methods in the development of computing systems has always attracted considerable attention. Formal methods have been a topic of research for many years, however they are rarely used in commercial contexts [31]. Even in some companies where formal methods have been employed, it is normally only to a limited extent and is often resisted by managers and technicians. This situation is hardly surprising since formal methods technology is largely perceived to consist of a collection of prototype notations and "research" tools which are difficult to use and do not scale up easily. There are many widely held misconceptions about the use of formal techniques [47]. It may be fair to say that formal methods research has to some extent been dominated by fundamental aspects rather than by problems in application [18].

It is important to realise that as the complexity of building computing systems is continually growing, a disciplined, systematic and rigorous methodology is essential for attaining a "reasonable" level of dependability and trust in these systems. In short, the use of formal methods is no substitute for good software production management, generally because formal methods use discrete mathematics to describe a system, logic proofs can be applied to ensure the correctness of the specifications of the system. More detailed reasons will be given in the following parts of the thesis. Time-critical systems may have the most to gain from the use of formal methods, such techniques are in fact useful in a wide variety of application areas in industry. Some examples show that formal development techniques together with their associated verification tools have been successfully applied in industry [10]. For example, assertional methods, temporal logic, process algebra and automata, have all been used with some degree of success.

Rolls-Royce and Associates have been applying formal methods (mainly VDM) for the development of software for time-critical systems, and nuclear power plants in particular, for a number of years [51, 52].

A number of medical instruments, which have life-critical functionality, are other typical examples of time-critical systems. For example, two Hewlett-Packard (HP) divisions have used formal specification in order to enhance the quality of cardiac care products [75, 32] by using HP-SL, a formal specification language based on VDM, developed at HP laboratories [12].

More recently, NASA commissioned work involving the application of formal methods to support digital flight control systems (DFCS) [111, 117]. Another example is the formal specification of the TCAS collision avoidance system, which was undertaken by N. Leveson et al. for the Federal Aviation Administration (FAA) [39].

In addition, formal specification languages and their semantics are themselves being standardised (e.g. LOTOS [62], VDM [19], and Z [100]). An important trigger for exploitation of research into formal methods comes from the interest of regulatory bodies or standardisation committees (e.g. the International Electro-technical Commission [60, 61], the European Space Agency [1], the MOD Defence Standard 00-55 [88], the MOD Defence Standard 00-56 [87], and the UK Railway Industry Association [8]).

For a formal methodology to be complete it must be able to fulfil the following requirements [49]:

1) *Specification.* It must be possible to state what a program is meant to do in a formal precise way, i.e. a specification in mathematical manner, unambiguously, consistent and complete.

2) *Verification.* Given the specification and a program obtained, it should be possible to prove using formal mathematical methods that the program does what the specification states. Therefore, if verification can provide proof that the program achieves its purpose, then it becomes the desired replacement or partial replacement for exhaustive testing and it can be called *proof of correctness*. It is essential to employ a set of proof rules to show correctness of properties.

Formal methods can therefore be seen as covering two areas: *specification* and *development*. In order to carry out verification it is necessary to be able to formally represent the program itself; that is, an exact specification of the semantics or meaning of each programming language construct is required.

A formal specification is usually composed of five primary components, a semantic model, a specification language (notation), a verification system/refinement calculus, development guidelines and supporting tools.

Formal methods can be classified into the following five classes or types, i.e., *Model-based, Logic-based, Algebraic, Process Algebra* and *Net-based (Graphical)* methods [85].

Model-based Approach. A system is modelled by explicitly giving definitions of states and operations that transform the system from one state to another. Examples include **Z** [116] and Vienna Development Method (**VDM**) [15].

Logic-based Approach. In which logics are used to describe the desired system properties, including low-level specification, temporal and probabilistic behaviours. Examples of logics include **Hoare Logic** [53, 54, 55] and **Temporal Logic** [110, 27, 93, 94, 26, 137].

Algebraic Approach. In this approach, an explicit definition of operations is given by relating the behaviour of different operations without defining states. There is no explicit representation of concurrency. Examples include **OBJ**[43] and **LARCH** [45].

Process Algebra Approach. Process algebras give an explicit model of concurrent processes, representing behaviours by means of constraints on allowable and observable communication between the processes. Examples include, *Communicating Sequential Processes (CSP)* [56], *Calculus of Communicating Systems (CCS)* [86], and *Language Of Temporal Ordering Specification (LOTOS)* [79].

Net-based Approach. Net-based approaches give an implicit concurrent model of the system in terms of (casual) data flow through a network, including representing conditions under which data can flow from one node in the net to another. Examples include **Petri Net** [109] and predicate transition nets [41].

These formalisms were used with real time context by conservative extensions to include time, such as Real-time Hoare Logic [57], Timed CCS (TCCS) [37], Timed CSP (TCSP) [108], and Timed Communicating Object Z (TCOZ) [80].

Methods and their supporting tools for the formal verification and analysis of system exist. For example, HyTech [50] is a symbolic model checker for linear hybrid automata [3], a subclass of hybrid automata that can be analysed automatically by computing with polyhedral state sets. A key feature of HyTech is its ability to perform parametric analysis, i.e. to determine the values of design parameters for which a linear hybrid automaton satisfies a temporal-logic requirement. HyTech has been most successful when applied to systems that involve an intricate interplay between discrete and continuous dynamics.

Mok et. al (within the SARTOR project) [90] developed a technique based on a logic for real-time systems (Real Time Logic (RTL)) and a specification language (Modechart). Real Time Logic first appeared in [65], and was inspired by Harel's statechart [48]. A method for verifying properties of systems specified in modechart was described in [66].

Both HyTech and Modechart are only suitable for the formal verification process during development. They cannot handle source code-level analysis against given properties. In addition, both formalisms are not compositional, which makes them hard for large-scale system evolution. The recent work on real-time constraints monitoring using RTL [64] and interval model checking [24] is based on Linear Time Logic (LTL). Although these analysers and their underlying logic are suitable for expressing real-time properties they both are non-compositional and not able to handle source code-level analysis. As we will see later on our tool is able to perform the necessary formal analysis at source code-level.

Analysers based on Anna [113] and PLEASE [126] are amongst early developed analysers. Anna is a language extension of Ada to include facilities for the formal specification of the intended behaviour of Ada programs. It augments Ada with precise machine-processable annotations so that well established formal methods of specification and documentation can be applied to Ada programs. Like Anna, PLEASE allows software to be annotated with formulae written in predicate logic; annotations can be used in proofs of correctness and to generate run-time assertion checks. As the logic in PLEASE is restricted to Horn clauses, specifications can be also transformed into prototypes which use Prolog to 'execute' pre- and post-conditions. Anna and PLEASE, however, do not deal with timing properties and are not compositional.

Temporal Rover [35] is a tool for the specification and verification/validation of protocols and reactive systems. It can automate the verification of real-time and relative temporal properties. The formal specification is written using a combination of Temporal Logic [110] and language of choice, such as C and Java. Temporal-logic assertions are inserted into the body of executable code in conjunction with pieces of formal specification. Temporal-logic assertions can be simulated using the Temporal-Rover simulator. However, Temporal Rover verifies the systems based on simulation and is not compositional as well. It offers simple pre and post conditions that is not enough for handling complex parallelism. The tool has no graphical output yet.

2.5 Software Evolution

Macroscopically, the role of computer software has undergone significant changes over a time span of little more than 50 years. Dramatic improvements in hardware performance, profound changes in computing architectures, vast increases in memory and storage capacity and a wide variety of exotic input and output options have all precipitated more sophisticated and complex computer-based systems [5].

Microscopically, it is safe to say that from the day that a large software system goes into service, functional, performance, operator and environmental requirements will undergo changes. Moreover, the delivered software system will contain some latent defects that were not detected during testing. These factors cause software systems inevitably to evolve in scale, environment and functionality, especially those successful enough to survive a long period [13, 7]. There is growing recognition that software, like all complex systems, evolves over a period of time [42]. It's essential for time-critical systems that both functional requirements and timing requirements are satisfied after a step in their evolution.

Taxonomy of Software Evolution

Software evolution consists of the activities required to keep a good and useful software system. They are *correcting defects, enhancing functionality and improving quality*. Depending on these activities, software evolution can be divided into:

- *corrective actions to fix latent defects*

  Defects refer to the system not performing as originally intended, or as specified in the requirements [7]. Some of examples include:

  - correcting a program that aborts, or
  - correcting a program that produces the incorrect results, or
  - correcting a program that doesn't produce the required results within the correct time.

- *adaptive actions to deal with changing environments*

  It includes systems changes, additions, insertions, deletions, modification, extensions, and enhancements to meet the evolving environment in which the systems must operate [7], e.g. it is necessary to modify some time parameters to fit new hardware configurations in real-time systems.

- *perfective actions to accommodate new requirements*

  This will happen as a consequence of a change in user requirements of the software. For example, a payroll suite may need to be altered to reflect new taxation laws; a real-time power station control system may need upgrading to meet new safety standards; a mail sorting system may need tuning to reduce total process time.

Software evolution is a process where maintenance is conducted continuously. For example, in a typical software evolution environment, software maintenance consumes about 90% of all resources. Furthermore, a number of theories, approaches and tools for software maintenance have been employed in the software evolution environment. For example, Gallagher's *program slicing* [40], dependency analysis and its tool set [129], and *traceability analysis* [17] are widely used to analyse change of impact in software evolution.

2.5.1 Managing Evolution of Time-critical Systems

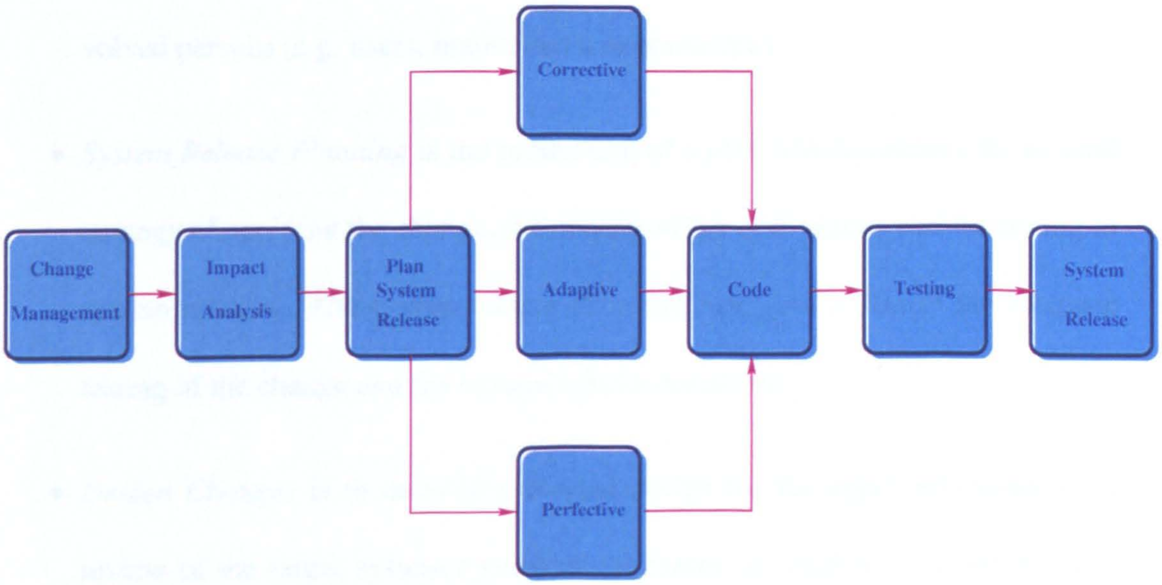


Figure 2.1: A Typical Software Evolution Process

There have been various development techniques for time-critical application. These were based on the traditional methods such as the waterfall model [105], spiral model [16], and incremental model [84]. However, other specialised techniques have emerged that utilise notions, such as object-oriented and component paradigms (for example, HRT-HOOD [29]).

Software evolution consists of the key activity, *handling changes*.

A typical software evolution process is shown in Figure 2.1 [7].

- *Change Management* is to uniquely identify, describe, and track the status of each requested change. In fact, change management is an ongoing activity throughout the software evolution process.

- *Impact Analysis* determines the scope of the requested change as a basis for its planning and implementation. It evaluates change for potential impacts on existing systems, other systems, documentation, hardware, data structure, and involved persons (e.g. users, maintainers, or operators.).

- *System Release Planning* is the production of a plan which contains the general strategy of applying the change, the content of the new system and the timing of system releases. The system release plan may also give a plan of the adequate testing of the change and the system release document.

- *Design Changes* is to develop a revised design for the approved changes. A review of the target system's structure, program, or modules will be given in this stage. System and program design is covered for each of the three activity categories of evolution, namely, *corrective, adaptive*, and *perfective*.

- *Coding* is to change the software to reflect the approved changes represented in the designs. The major activities are:
  - implement and review all changes to code, Temporal Rover [35] is a tool for the specification and verification/validation of protocols and reactive systems. It can automate the verification of real-time and relative temporal properties. The formal specification is written using a combination of Temporal Logic [110] and language of choice, such as C and Java. Temporal-logic assertions are inserted into the body of executable code in conjunction with pieces of formal specification. Temporal-logic assertions can be simulated using the Temporal-Rover simulator. However, Temporal Rover verifies the systems based on simulation and is not compositional as well. It offers simple pre and post conditions that is not enough for handling complex parallelism. The tool has no graphical output yet.
  - restore or place the source code under control of the configuration management system, and
  - update the change request to reflect the modules or units changed.
- *Testing* is to ensure compliance with the original requirements and the approved changes. The major testing activities are:
  1. *Human testing*: requirements, design, and code walk-throughs or inspections.
  2. *Computer testing*:

  * *Unit test*: all code changes by module or unit.
  * *Integration test*: the interfaces between each module of the program and the program as a whole.
  * *System test*: the interfaces between programs to ensure that the system meets all of the original requirements plus the added changes.
  * *Acceptance testing*: where the user approves the revised system.

- *System Release* is to deliver the system and updated documentation to users for installation and operation.

Steps in the Evolution Process	Current Usage	Formal Methods Usage
Change Analysis	informal/structured mature techniques	structured non-well-developed
Impact Analysis	ad hoc	mathematical analysis
Change Description	informal/structured	formal specification
New System Design	informal/structured e.g. pseudo-code technique	formal specification and verification
Coding	restructuring code directly	refinement + derive code from concrete specification
Testing	mature testing methods and tools	Test suites generated by using Formal Specification, few practical tools available

Table 2.1: Formal Methods Usage

Formal techniques can be utilised within the above process. In Table 2.1, we summarise such usage. We develop compositional techniques to illustrate such utilisation in the thesis.

2.6 Summary

Time-critical systems are these in which their "correctness" depends on both functional and timing correctness. Using formal methods is crucial in the specification, development, and implementation of these systems. The chapter reviewed current techniques for the "evolution" with the need for compositional and sound methodology for handling that evolution.

Chapter 3

Preliminaries

Objectives:

To give a computational model

To introduce formal basis: Interval Temporal Logic (ITL)

To give a short review of Temporal Logics

To describe Tempura, an executable subset of ITL

To introduce Refinement Calculus

To give the Assumption/Commitment paradigm

3.1 Introduction

We begin by establishing a computation model which is suitable for modelling time-critical systems. Then we describe a formal basis of our methodology, namely, Interval Temporal Logic (ITL). ITL forms a specification oriented methodology for the engineering of time-critical systems. The choice of ITL is based on a number of reasons. These reasons will be presented in this chapter in conjunction with a number of comparisons between ITL and other similar logics. Furthermore, all refinement relations and rules, as well as the process of run-time analysis, are precisely formulated in ITL. ITL has an executable subset, namely, Tempura. It can be used for simulation and rapid prototyping purposes. A refinement calculus of ITL will then be introduced in this chapter. In addition, as a key issue of preliminaries, the *assumption/commitment* paradigm will be discussed.

3.2 Computational Model

First of all, we develop a computational model which attempts to overcome some of the simplifying assumptions, such as, "all processes are periodic"; "the processes are completely independent of each other"; or "all processes have a fixed worst-case execution time", made both by other formal real-time languages and by other scheduling theories. A time-critical application within our model is characterised by:

- A fixed set of periodic and sporadic tasks. Each periodic task has an associated deadline. Each sporadic task has a defined minimum inter-arrival time and a deadline.
- Computation times may be between a minimum (best-case) and a maximum (worst-case).
- Communication between tasks is asynchronous. Tasks are related to one another by precedence relationships.
- Tasks may execute on a single processor, a multiprocessor or a distributed system.

  - Tasks allocated to a single processor may be preempted at run-time by other tasks.
  - Multi-node systems with general static process allocation are calculated pre-run-time.

- Processors are either connected by a broadcast local area network (token ring or token passing bus) or a point-to-point architecture.

The main advantage of this model in conjunction with the formal development method is that it allows us to remove a number of significant restrictions found in other approaches, for example, the maximal parallelism hypothesis, which is an unrealistic assumption made in languages such as Timed-CSP [114].

Computation

We take a view that a unit of computation defines mathematically an abstract architecture upon which applications will execute. A *system* is a collection of *agents*, which is our unit of computation. Systems can themselves be viewed as single agents and composed into larger systems. At any instant in time a system can be thought of as having an unique *state*. The system state is defined by the state variables of the system and, for a concurrent system, by the values in the communication links.

Computation is defined as any process that results in a change of system state.

An agent is described by a set of computations which may transform a private data-space and may read and write to communication links during execution. Each computation may have both minimum and maximum execution times imposed.

A system should be tightly related to its *environment*. A system should realise correct computations to meet requirements from its environment and supply feedback to the environment. The environment of a system embraces all external factors or forces, which include surrounding entities, conditions or influences, especially affecting the existence or development of the system. For example, for a robot control system, its environment includes all sensors and actuators of the robot.

Behaviours and Properties

A *behaviour* in our model is defined as a sequence of states, i.e., an interval σ. Hence, a behaviour could be finite or infinite. A behaviour is also the same as a unit of computation. A behaviour is called *full* behaviour if it contains all the state variables of the system, otherwise, it is called *partial*. A partial behaviour can be obtained by hiding some state variables (formally, it is a projected behaviour over certain state variables).

A property, P, is a set of behaviours. A general classification of properties is readily available:

safety (*something bad does not happen*) and

liveness (*something good will eventually happen*) property.

A safety property is finitely refutable, so if a behaviour does not satisfy the property, then we can find who took the step that violated it. A property P is a liveness property if and only if every finite behaviour with the prefix σ is a prefix of a behaviour in P.

3.3 Interval Temporal Logic

3.3.1 Introduction

Interval Temporal Logic (ITL) is an extension of classical first order logic especially designed for representing time-dependent behaviour. It has proved to be an effective notation for specifying and reasoning about real time critical systems.

ITL is a flexible notation for both propositional and first order reasoning about periods of time found in descriptions of hardware and software systems. It can handle both sequential and parallel composition unlike most temporal logics. It offers powerful and extensible specification and proof techniques for reasoning about properties involving safety, liveness and timeliness.

3.3.2 Temporal Logic Vs Interval Temporal logic

Temporal logic has its origins in philosophy, where it was used to analyse the structure or topology of time. In recent years, it was found to be valuable in real-time applications.

In physics and mathematics, time has traditionally been represented as just another variable. First order predicate calculus is used to reason about expressions containing the time variable, and there is thus apparently no need for a special *temporal* logic.

However, philosophers found it useful to introduce special temporal operators, such as $\Box$ (henceforth) and $\Diamond$ (eventually), for the analysis of temporal connectives in languages. The new formalism was soon seen as a potentially valuable tool for analysing the topology of time. Various types of semantics can be given to the temporal operators depending on whether time is linear, parallel or branching. Another aspect is whether time is discrete or continuous [82].

Temporal logic is *state-based.* A structure of states is the key concept that makes temporal logic suitable for system specification. Mainly, the types of temporal semantics include [82] *interval semantics, point semantics, linear semantics, branching semantics* and *partial order semantics.*

The various temporal logics can be used to reason about *qualitative* temporal properties. Safety properties that can be specified include mutual exclusion and absence of deadlock. Liveness properties include termination and responsiveness. Fairness properties include scheduling a given process infinitely often, or requiring that a continuously enabled transition ultimately fires. Various proof systems and decision procedures for finite state systems can be used to check the correctness of a program or system.

In real-time temporal logics, quantitative properties can also be expressed such as periodicity, real-time response (deadline), and delays. Early approaches to real-time temporal logics were reported by Ostroff [101] and Benveniste [14]. Since then, real-time logics have been explored in great detail.

In this thesis, Interval Temporal Logic (ITL) was chosen as formal preliminaries. ITL was originally developed by Ben Moszkowski in order to model digital circuits [92]. Later it was designed particularly as a formalism for the specification and design of software systems [93, 95, 96, 27]. Timing constraints are expressible and furthermore most imperative programming constructs can be viewed as formulas in a slightly modified version of ITL by Zedan and Cau [26].

While other temporal logics, such as Discrete Temporal Logics (PTL) [36], based on the notions of points, ITL is concerned with the truth of statements over *intervals*, rather than just to a point in time; that is, the starting and ending points are both considered. However, its syntax contains the basic temporal operators, which are the same as of PTL, such as $\bigcirc(next), \Diamond(sometime)$, and $\Box(always)$. Except for shared advantages with other temporal logics, the reasons of choosing ITL result from its own characteristics, which are presented as follows.

- ITL is a flexible notation for both propositional and first-order reasoning about periods of time found in descriptions of hardware and software systems.

- Unlike most temporal logics, ITL can handle both sequential and parallel composition and offer powerful and extensible specification and proof techniques for reasoning about properties involving safety, liveness and time.

- Tempura [94], an executable subset of ITL, provides an executable framework for developing, analysing and experimenting with suitable ITL specifications.

- ITL has a complete axiomatic system [97].

- In addition, Zedan and Cau have provided a refinement calculus for ITL [26] that can "translate" an ITL formula into executable code.

3.3.3 Syntax and Semantics

The key notion of ITL is an *interval*. An interval σ is considered to be a (in)finite sequence of states σ_0, $\sigma_1 \ldots$, where a state σ_i is a mapping from the set of variables Var to the set of values Val. The length $|\sigma|$ of an interval $\sigma_0 \ldots \sigma_n$ is equal to n (one less than the number of states in the interval[1], i.e., a one state interval has length 0). The notation $\sigma_{i:j}$ denotes the subinterval of σ of length $j - i$ with states $\sigma_i, \sigma_{i+1}, ..., \sigma_j$.

Table 3.1: Syntax of ITL

	Expressions
$e ::=$	$\mu \mid a \mid A \mid g(exp_1, \ldots, exp_n) \mid \imath a \colon f$
	Formulae
$f ::=$	$p(exp_1, \ldots, exp_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \bullet f \mid \mathsf{skip} \mid f_1 \,;\, f_2 \mid f^*$

The syntax of ITL is defined in Table 3.1 where μ is an integer value, a is a static variable (doesn't change within an interval), A is a state variable (can change within an interval), v a static or state variable, g is a function symbol and p is a predicate symbol.

The informal semantics of the most interesting constructs are as follows:

- $\imath a \colon f$: choose a value of a such that f holds. If there is no such an a then $\imath a \colon f$ takes an arbitrary value from a's range.
- skip: unit interval (length 1).
- $f_1 \,;\, f_2$: holds if the interval can be decomposed ("chopped") into a prefix and suffix interval, such that f_1 holds over the prefix and f_2 over the suffix, or if the interval is infinite and f_1 holds for that interval.

[1]This has always been a convention in ITL

- f^*: holds if the interval is decomposable into a finite number of intervals such that for each of them f holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which f holds.

Table 3.2: Frequently used abbreviations

$\bigcirc f$	$\widehat{=}$ skip ; f	next
more	$\widehat{=} \bigcirc$*true*	non-empty interval
empty	$\widehat{=} \neg$*more*	empty interval
inf	$\widehat{=}$ *true* ; *false*	infinite interval
finite	$\widehat{=} \neg$*inf*	finite interval
$\Diamond f$	$\widehat{=}$ *finite* ; f	sometimes
$\Box f$	$\widehat{=} \neg\Diamond\neg f$	always
if f_0 then f_1 else f_2	$\widehat{=} (f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2)$	if then else
$f_0 \| f_1$	$\widehat{=} f_0 \wedge f_1$	parallel composition
fin f	$\widehat{=} \Box$(empty $\supset f$)	final state
$\diamondsuit f$	$\widehat{=}$ *finite* ; f ; *true*	some subinterval
$\boxed{a} f$	$\widehat{=} \neg(\diamondsuit \neg f)$	all subintervals
keep f	$\widehat{=} \boxed{a}$(skip $\supset f$)	all unit subintervals
while f_0 do f_1	$\widehat{=} (f_0 \wedge f_1)^* \wedge$ *fin* $\neg f_0$	while loop
$\bigcirc exp$	$\widehat{=} \imath a$: $\bigcirc(exp = a)$	next value
fin exp	$\widehat{=} \imath a$:*fin* $(exp = a)$	end value
$A := exp$	$\widehat{=} \bigcirc A = exp$	assignment
$exp_1 \leftarrow exp_2$	$\widehat{=}$ *finite* $\wedge$ (*fin* exp_1) $= exp_2$	temporal assignment
exp_1 *gets* exp_2	$\widehat{=}$ *keep* $(exp_1 \leftarrow exp_2)$	gets
stable exp	$\widehat{=}$ exp *gets* exp	stability

These constructs enables us to define programming constructs, including $\Diamond$(sometimes), $\Box$(always), $\bigcirc$(next), etc. Table 3.2 contains some frequently used abbreviations. For example, a simple heating controller program can be defined it as:

$$\begin{aligned} Tempurature\_Controller \;\widehat{=}\;& (if\ temp > Threshold \\ & then\ \bigcirc(Heater = ON) \\ & else\ \bigcirc(Heater = OFF))^{*} \end{aligned}$$

The following are some examples illustrating ITL:

1. In an interval, the variable I at *some time* equals 1 and at *some later time* equals 2 can be expressed as:

$$\Diamond[(I = 1) \wedge \Diamond(I = 2)]$$

2. In an interval, if the variable I *always* equals 1 and in the next state the variable J equals 2 then it follows that the expression I + J equals 3 in the next state:

$$[\Box(I = 1) \wedge \bigcirc(J = 2)] \quad \supset \quad \bigcirc(I + J = 3)$$

3. The formula

$$(K + 1 \rightarrow K)\,;(K + 2 \rightarrow K)$$

is true in an interval if, and only if, that interval can be chopped into two sub-intervals such that the sub-formula $K + 1 \rightarrow K$ is true on the first subinterval and the sub-formula $K + 2 \rightarrow K$ is true on the second subinterval. The net effect is that K increases by 3. This is expressed by the following property:

$$[(K+1 \rightarrow K)\,;(K+2 \rightarrow K)] \supset (K+3 \rightarrow K)$$

The formal semantics of ITL is as follows: Assume $\mathcal{X}$ be a choice function which maps any nonempty set to some element in the set. It is written $\sigma \sim_v \sigma'$ if the intervals σ and σ' are identical with the possible exception of their mappings for the variable v. It is assumed a fixed interpretation $\mathcal{I}$ which serves two purposes. First, it associates data domains $\mathcal{I}_1, \mathcal{I}_2, ...,$ with the corresponding sorts 1, 2, ... Secondly, $\mathcal{I}$ gives meaning to the predicate and function symbols. More precisely, $\mathcal{I}$ maps each n-ary predicate symbol p to an n-ary relation $\mathcal{I}(p) \in 2^{\mathcal{I}_{\hat{p}_1} \times \cdots \times \mathcal{I}_{\hat{p}_n}}$. Similarly, each n-ary function symbol f is associated with a n-ary function $\mathcal{I}(f) \in \mathcal{I}_{\hat{f}_1} \times ... \times \mathcal{I}_{\hat{f}_n} \rightarrow \mathcal{I}_{\hat{f}_{n+1}}$ that suits f's sort requirements. It is assumed that $\mathcal{I}$ contains interpretations for the arithmetic operators and relations for natural numbers as well as operators for manipulating finite lists (e.g., subscripting and list length).

The meaning of an expression is defined inductively:

- Static or state variable:

  $\mathcal{E}_\sigma[\![v]\!] = \sigma_0(v)$.

  The value of a variable for an interval σ is the variable's value in the initial state σ_0.

- Function: $\mathcal{E}_\sigma[\![f(exp_1, ..., exp_n)]\!] = \mathcal{I}(f)(\mathcal{E}_\sigma[\![exp_1]\!] ..., \mathcal{E}_\sigma[\![exp_n]\!]$.

- Definite descriptions: $\mathcal{E}_\sigma[\![\imath v : f]\!] = \begin{cases} \mathcal{X}(u) \ if \ u \neq \{\} \\ \mathcal{X}(\mathcal{I}_{\hat{v}}) \ otherwise, \end{cases}$

  where u is the set of values of the static variable v in the interval σ' such that $\sigma \sim_v \sigma'$ and $\mathcal{M}_{\sigma'}[\![f]\!] = true$:

$$u = \{\sigma'(v) : \sigma' \in Int, \sigma \sim_v \sigma' \ and \ \mathcal{M}_{\sigma'}[\![f]\!] = true\}.$$

  If u is empty, the description equals some value selected from v 's domain $\mathcal{I}_{\hat{v}}$. Since v is static, it has a unique value in σ' denoted here $\sigma'(v)$.

The meaning of formulas is defined as:

- Predicates: $\mathcal{M}_\sigma[\![p(exp_1, \ldots, exp_n)]\!] =$ true iff $\langle \mathcal{E}_\sigma[\![exp_1]\!], \ldots, \mathcal{E}_\sigma[\![exp_n]\!] \rangle \in \mathcal{I}(p)$.

- Negation: $\mathcal{M}_\sigma[\![\neg f]\!] =$ true iff $\mathcal{M}_\sigma[\![f]\!] =$ false.

- Conjunction: $\mathcal{M}_\sigma[\![f_1 \wedge f_2]\!] =$ true iff $\mathcal{M}_\sigma[\![f_1]\!] =$ true and $\mathcal{M}_\sigma[\![f_2]\!] =$ true.

- Universal quantification: $\mathcal{M}_\sigma[\![\forall v \bullet f]\!] =$ true

  iff for all σ' s.t. $\sigma \sim_v \sigma'$, $\mathcal{M}_{\sigma'}[\![f]\!] =$ true,

  for all intervals σ' that are identical to σ except possibly for the behaviour of the variable v (i.e., $\sigma \sim_v \sigma'$).

- Unit interval: $\mathcal{M}_\sigma[\![\textsf{skip}]\!] =$ tt iff $|\sigma| = 1$.

- Chop: $\mathcal{M}_\sigma[\![f_1 \ ; \ f_2]\!] =$ true iff

  (exists a k, s.t.$\mathcal{M}_{\sigma_0 \ldots \sigma_k}[\![f_1]\!] =$ true and

((σ is infinite and $\mathcal{M}_{\sigma_k \ldots}[\![f_2]\!] = \text{true}$) or

(σ is finite and $k \leq |\sigma|$ and $\mathcal{M}_{\sigma_k \ldots \sigma_{|\sigma|}}[\![f_2]\!] = \text{true}$)))

or (σ is infinite and $\mathcal{M}_\sigma[f_1]$).

- Chop-star: $\mathcal{M}_\sigma[f^*] = \text{true}\ iff$

  if σ is infinite then

  (exist $l_0, \ldots, l_n$ s.t. $l_0 = 0$ and

  $\mathcal{M}_{\sigma_{l_n} \ldots}[\![f]\!] = \text{true}$ and

  for all $0 \leq i < n$, $l_i \leq l_{i+1}$ and $\mathcal{M}_{\sigma_{l_i} \ldots \sigma_{l_{i+1}}}[\![f]\!] = \text{tt}$.)

  or

  (exist an infinite number of l_i s.t. $l_0 = 0$ and

  for all $0 \leq i$, $l_i \leq l_{i+1}$ and $\mathcal{M}_{\sigma_{l_i} \ldots \sigma_{l_{i+1}}}[\![f]\!] = \text{tt}$.)

  else

  (exist $l_0, \ldots, l_n$ s.t. $l_0 = 0$ and $l_n = |\sigma|$ and

  for all $0 \leq i < n$, $l_i \leq l_{i+1}$ and $\mathcal{M}_{\sigma_{l_i} \ldots \sigma_{l_{i+1}}}[\![f]\!] = \text{tt}$.)

Within above formulas, the constants "*true*" and "*false*" are the interval forms of the classical truth values **T** and **F**. Formally, they can be defined as:

$$false \mathrel{\widehat{=}} f \wedge \neg f \ \ for\ any\ formula\ f;$$

$$true \mathrel{\widehat{=}} \neg false$$

3.3.4 Types

There are two basic builtin types in ITL (which can be given pure set-theoretic definitions). These are integers $\mathcal{N}$ (together with standard relations of inequality and equality) and Boolean (*true* and *false*).

Further types can be built from these by means of $\times$ and the power set operator $\mathcal{P}$ (in a similar fashion as adopted in the specification language Z).

For example, the following introduces a variable x of type T

$$(\exists x : T) \cdot f \;\widehat{=}\; \exists x \cdot (type(x, T) \wedge f)$$

Here $type(x, T)$ denotes a formula describing the desired type of x. For example, $type(x, T)$ could be $0 \leq x \leq 7$ and so on. Although this might seem to be a rather inexpressive type system, richer types can be added.

3.4 Assumption/Commitment Paradigm

Comparing to sequential programs, concurrent programs are much harder to specify and verify. *Assumption/commitment* (sometimes also called *rely/guarantee* [133]), as against monolithic, specification paradigm has therefore been introduced [132]. The assumption/commitment paradigm has been advocated in numerous variations (see [123, 103]) for the specification of interactive components of real-time/distributed systems.

It provides a concept for the description of an interface [2] between a system and its environment [21]. The basic idea of the assumption/commitment paradigm is to make a clear separation in an interface specification of a component into the responsibilities of the component and those of its environment within their interaction [21]. In the assumption/commitment paradigm, a component will be verified to satisfy a commitment under the condition that the environment satisfies an assumption. In other words, it suggests the structuring of specifications into assumptions about the behaviour of the component's environment and into commitments that are fulfilled by the component provided the environment fulfils these assumptions.

The idea that specifications are conveniently formulated and manipulated in the form of assumption/commitment conditions is not new. Pre/postcondition specifications for sequential programs are special case of assumption/commitment specifications, in which the precondition expresses the conditions on the program variables the program assumes on when control enters it, and the postcondition expresses the conditions the program commits when and if control leaves it. Assumption/commitment rules were first studied as extensions of Hoare Logic. The so-called "assumption / commitment" method [89] introduced by Misra and Chandy in 1981 is suited to describe open systems, based on Hoare's pre/postcondition specifications [53]. But this method is unable to prove temporal properties. The Floyd/Hoare techniques for proving partial correctness of sequential programs [38, 53] can be viewed as a special case of the proof technique for assumption/commitment properties. [103, 68] studied mainly

[2]The logical connection between a real-time distributed (sub-)system and its environment is called its *interface*.

in the framework of state-based system models. Abadi and Lamport [2] studied assumption/commitment paradigms for achieving compositionality for specification and verification techniques using the Temporal Logic of Actions (TLA).

An assumption/commitment specification for a system Sys is a specification of the basic form $A \supset C$, where A is an *assumption* condition and C is a *commitment* condition. An *assumption* condition expresses the conditions that Sys assumes its environment to provide, and a *commitment* condition expresses what Sys commits to provide in return. Formally, for a system Sys, the assumption/commitment style specification can be expressed in ITL as follows:

$$w \wedge As \wedge Sys \supset Co \wedge \mathit{fin}\ w',$$

which states that if the state formula w is true in the initial state and the assumption As is true over the interval in which Sys is operating, then the commitment Co is also achieved. Furthermore the state formula w' is true in the interval's final state or is vacuously true if the interval does not terminate. This is particularly important as As could be a formula asserting various assumptions about the environment in which the system, under consideration, is operating. For convenience, in some cases, it can abbreviate $(As \wedge w)$ to Ass and $(Co \wedge \mathit{fin}\ w')$ to Com.

The following is an example adapted from the mine pump control problem [74, 22, 81, 70]:

Water percolating into a mine is pumped out of the mine. A pump controller switches the pump, depending on the methane level, i.e. to avoid the risk of explosion, the pump must be switched off when the methane level is above a critical level. The presence of methane is measured in units of Pascal and indicated by a methane sensor as a value of $Pressure$ (a real number). There is a critical level, $MineExplo$, above which the pump must be switched off in 1 second. The system, $PumpController$ can be expressed as:

$$\begin{array}{l}
(Ass, Com), \quad where \\
Ass: \ true \\
Com: \ \Box\ (\\
(Pressure \geqslant MineExplo \ \supset \ \Diamond(Pump = OFF)) \wedge \\
(Pressure < MineExplo \ \supset \ \Diamond(Pump = ON)))
\end{array}$$

The total specification means that the pump controller, $PumpController$ guarantees that it eventually turns the mine pump off ($Pump = OFF$), assuming that the methane level reaches or exceeds the critical level, and eventually keeps the mine pump operating ($Pump = ON$), assuming that the methane level is below the critical level.

Using such an approach to describe the desired properties we can achieve compositional analysis and validation, which will be described later on.

3.5 Refinement Calculus

A refinement calculus for ITL is readily available based on Back [9] and Morgan's [91] work. Using such a calculus, an ITL formula could be refined into concrete code written in languages such as Ada, C. Especially, the development of an executable subset of ITL, known as Tempura [94], was a milestone in the use of ITL as it enables one to check, debug and simulate the design [26]. Therefore, the design and implementation can be simply done in Tempura.

Refinement laws will be applied to transform from the abstract level to the concrete level. During this transition, both abstract and concrete representations are allowed to intermix. The representation at the abstract level is done using solely pure ITL primitives. Firstly, the refinement ordering relation, "$\sqsubseteq$", is defined in the normal way as:

$$P \sqsubseteq Q \mathrel{\widehat{=}} Q \supset P.$$

Clearly, "$\sqsubseteq$" is a partial order. As usual then,

- a sequence P_k of agents is called increasing if P_k gets progressively stronger

$$\text{for all } k \ P_k \sqsubseteq P_{k+1}$$

- a sequence P_k of agents is called decreasing if P_k gets progressively weaker

$$\text{for all } k \ P_{k+1} \sqsubseteq P_k$$

It is elementary fact that each

- increasing sequence has a limit which is the weakest agent stronger than each P_k (namely, $\bigcap_k P_k$);
- decreasing sequence has a limit which is the strongest agent weaker than each P_k (namely, $\bigcup_k P_k$).

Let $\mathcal{P}$ be the set of all agents. A function F from $\mathcal{P}$ to $\mathcal{P}$

- is $\wedge$-continuous if for every increasing chain P_k: $F(\bigcap_k P_k) = \bigcap_k F(P_k)$,
- is $\vee$-continuous if for every decreasing chain P_k: $F(\bigcup_k P_k) = \bigcup_k F(P_k)$,

  and
- is monotonic if $P \sqsubseteq Q \quad \supset \quad F(P) \sqsubseteq F(Q)$.

Assignment

The assignment is introduced with the following law

$$(:= -1) \quad x := exp \qquad \equiv \qquad \bigcirc x = exp$$

Non-deterministic choice

Let P and Q be two agents, $P \vee Q$ denotes an agent that behaves either as P or Q, but does not determine which one. Hence the environment cannot control or predict the result. The following are some basic laws governing $\vee$.

The choice between the same agents is vacuous.

$$(\vee -1)\ P \vee P \equiv P$$

The choice is commutative and associative:

$$(\vee -2)\ P \vee Q \equiv Q \vee P$$

$$(\vee -3)\ P \vee (Q \vee R) \equiv (P \vee Q) \vee R$$

where R denotes an agent.

The choice has *true* as its zero.

$$(\vee -4)\ \mathit{true} \vee P \equiv \mathit{true}$$

Note: for agent P and Q we have

$$P \sqsubseteq Q \ \equiv\ (P \vee Q) \equiv P$$

If then else–conditional

The conditional is both idempotent and associative.

$$(if\ -1)\ if\ f_0\ then\ f\ else\ f\ \equiv\ f$$

$$(if\ -2)\ if\ f_0\ then\ f_1\ else\ (if\ f_0\ then\ f_2\ else\ f_3)$$

$$\equiv\ if\ f_0\ then\ f_1\ else\ f_3$$

$$\equiv\ if\ f_0\ then(if\ f_0\ then\ f_1\ else\ f_2)\ else\ f_3.$$

The following two laws describe how conditional makes a choice between its arguments.

$$(if\ -3)\ if\ true\ then\ f_1\ else\ f_2\ \equiv\ f_1$$

$$(if\ -4)\ if\ f_0\ then\ f_1\ else\ f_2\ \equiv\ if\ \neg f_0\ then\ f_2\ else\ f_1.$$

The relationship between conditional and $\vee$ is given by:

$$(if\ -5)\ if\ f_0\ then\ (f_1 \vee f_2)\ else\ f_3\ \equiv\ (if\ f_0\ then\ f_1\ else\ f_3)\ \vee\ (if\ f_0\ then\ f_2\ else\ f_3)$$

$$(if\ -6)\ (if\ f_0\ then\ f_1\ else\ f_2)\ \vee\ f_3\ \equiv\ if\ f_0\ then(f_1 \vee f_3)\ else\ (f_2 \vee f_3).$$

To allow unnesting of conditionals, there is:

$$(if\ -7)\ if\ f_{00}\ then\ (if\ f_{01}\ then\ f_1\ else\ f_2)\ else\ (if\ f_{02}\ then\ f_1\ else\ f_2)$$

$$\equiv\ if\ (if\ f_{00}\ then\ f_{01}\ else\ f_{02})\ then\ f_1\ else\ f_2$$

Chop–sequential composition

The following rules describes the characteristics of ';'.

';' has empty as a unit and is associative.

$$(; -1)\quad \mathsf{empty} ; f \equiv f \equiv f ; \mathsf{empty}$$

$$(; -2)\quad (f_1 ; f_2) ; f_3 \equiv f_1 ; (f_2 ; f_3)$$

The chop operator distributes over nondeterministic choice and conditional.

$$(; -3)\quad f_1 ; (f_2 \vee f_3) ; f_4 \equiv (f_1 ; f_2 ; f_4) \vee (f_1 ; f_3 ; f_4)$$

$$(; -4)\quad (\mathsf{if}\ f_0\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2) ; f_3 \equiv \mathsf{if}\ f_0\ \mathsf{then}\ (f_1 ; f_3)\ \mathsf{else}\ (f_2 ; f_3)$$

The chop operator is $\vee$-continuous.

$$(; -5)\quad f_1 ; \bigcup_i g_i \equiv \bigcup_i f_1 ; g_i$$

While loop

The following law introduces the *while* loop.

$$(\mathsf{while}\ -1)\quad \mathsf{while}\ f_0\ \mathsf{do}\ f_1 \equiv (f_0 \wedge f_1)^* \wedge \mathit{fin}\ \neg f_0$$

The following law is for the introduction of a non-terminating loop

$$(\text{while} -2) \quad \text{while } \mathit{true} \text{ do } f_1 \quad \equiv \quad f_1^*$$

Parallel

The following are some laws for the parallel agent.

$$(\| -1) \quad f \parallel \mathit{true} \quad \equiv \quad f$$

$$(\| -2) \quad f_0 \parallel f_1 \quad \equiv \quad f_1 \parallel f_0$$

$$(\| -3) \quad f_0 \parallel (f_1 \vee f_2) \quad \equiv \quad (f_0 \parallel f_1) \vee (f_0 \parallel f_2)$$

$$(\| -4) \quad (f_0 \parallel f_1) \parallel f_2 \quad \equiv \quad f_0 \parallel (f_1 \parallel f_2)$$

$$(\| -5) \quad (\text{if } f_0 \text{ then } f_1 \text{ else } f_2) \parallel f_3 \quad \equiv \quad \text{if } f_0 \text{ then } (f_1 \parallel f_3) \text{ else } (f_2 \parallel f_3)$$

Variable introduction

The following is the local variable introduction law.

$$(\text{var} -1) \quad \text{var}\, x \text{ in } P \quad \equiv \quad \exists x \bullet P$$

where P is an agent.

An example with refinement The initial specification of the operator Control in ITL is extracted from a robot control system:

Let *l-o-c* and *r-o-c* denote respectively the left and right steering commands received from the operator. Let *ll-o-c* and *lr-o-c* denote respectively the last left and last right steering commands received from the operator.

The specification of operator control is then as follows:

$$
\begin{array}{l}
OCS \mathrel{\widehat{=}} \\
\exists \mathit{ll\text{-}o\text{-}c}, \mathit{lr\text{-}o\text{-}c} \bullet (\\
\quad \mathit{ll\text{-}o\text{-}c} = 0 \wedge \mathit{lr\text{-}o\text{-}c} = 0 \wedge \\
\quad (\mathit{o\text{-}act} = (\mathit{l\text{-}o\text{-}c} \neq \mathit{ll\text{-}o\text{-}c} \vee \mathit{r\text{-}o\text{-}c} \neq \mathit{lr\text{-}o\text{-}c}) \wedge \\
\quad\quad \bigcirc \mathit{ll\text{-}o\text{-}c} = \mathit{l\text{-}o\text{-}c} \wedge \bigcirc \mathit{lr\text{-}o\text{-}c} = \mathit{r\text{-}o\text{-}c} \\
\quad)^{*} \\
)
\end{array}
$$

First we introduce with (var -1) the local variables *ll-o-c* and *lr-o-c*:

$$
\begin{array}{l}
OCS \\
\sqsubseteq \\
\mathsf{var}\ \mathit{ll\text{-}o\text{-}c}, \mathit{lr\text{-}o\text{-}c}\ \mathsf{in}\ (\\
\quad \mathit{ll\text{-}o\text{-}c} = 0 \wedge \mathit{lr\text{-}o\text{-}c} = 0 \wedge \\
\quad (\mathit{o\text{-}act} = (\mathit{l\text{-}o\text{-}c} \neq \mathit{ll\text{-}o\text{-}c} \vee \mathit{r\text{-}o\text{-}c} \neq \mathit{lr\text{-}o\text{-}c}) \wedge \\
\quad\quad \bigcirc \mathit{ll\text{-}o\text{-}c} = \mathit{l\text{-}o\text{-}c} \wedge \bigcirc \mathit{lr\text{-}o\text{-}c} = \mathit{r\text{-}o\text{-}c} \\
\quad)^{*} \\
)
\end{array}
$$

Then we will refine * into a while loop using law (while -2).

```
$\sqsubseteq$

var $ll\text{-}o\text{-}c, lr\text{-}o\text{-}c$ in (
  $ll\text{-}o\text{-}c = 0 \wedge lr\text{-}o\text{-}c = 0 \wedge$
  while $true$ do (
    $o\text{-}act = (l\text{-}o\text{-}c \neq ll\text{-}o\text{-}c \vee r\text{-}o\text{-}c \neq lr\text{-}o\text{-}c) \wedge$
    $\bigcirc ll\text{-}o\text{-}c = l\text{-}o\text{-}c \wedge \bigcirc lr\text{-}o\text{-}c = r\text{-}o\text{-}c$
  )
)
```

Finally, the assignment statements are introduced with (:=-1):

```
$\sqsubseteq$

var $ll\text{-}o\text{-}c, lr\text{-}o\text{-}c$ in (
  $ll\text{-}o\text{-}c = 0 \wedge lr\text{-}o\text{-}c = 0 \wedge$
  while $true$ do (
    $o\text{-}act = (l\text{-}o\text{-}c \neq ll\text{-}o\text{-}c \vee r\text{-}o\text{-}c \neq lr\text{-}o\text{-}c) \wedge$
    $ll\text{-}o\text{-}c := l\text{-}o\text{-}c \wedge lr\text{-}o\text{-}c := r\text{-}o\text{-}c$
  )
)
```

3.6 Tempura

3.6.1 Introduction

An important reason of choosing ITL is the availability of an executable subset (known as Tempura) of the logic. Originally proposed by Ben Moszkowski [94], Tempura is an executable sub-language of ITL. The tempura program is deterministic, i.e. no arbitrary choices (either of computation length or variable assignment) can be made during execution. For example, neither the formula $\neg\mathsf{skip}$ nor the formula $(I = 0) \vee (I = 1)$ is executable, as both are non-deterministic. The former describes any interval of length other than one, the latter gives a choice of values for the variable I. It maintains the equivalence between program and logical interpretation. Its syntax resembles that of ITL. However, the syntax of Tempura is restricted in order to exclude formulae such as $\neg$ and $\vee$. This means that some operators of ITL, such as negation, cannot be defined at all in Tempura, and that some others can only be defined in a restricted form, such as variable assignment. It has as data-structures integers and booleans and list construct to built more complex ones.

Tempura offers a means for rapidly developing, testing and analysing suitable ITL specifications. As with ITL, Tempura can be extended to contain most imperative programming features and yet retain its distinct temporal feel. The use of ITL and Tempura combines the benefits of traditional proof methods balanced with the speed and convenience of computer-based testing through execution and simulation. The entire process can remain in one powerful logical and compositional framework.

3.6.2 Another ITL-based Executable Language

As well as Tempura, there is another temporal language, called Tokio, based on ITL. Tokio was proposed by Fujita and Moto-oka [4] for the description of computer hardware. It is based on ITL with influence from Tempura. It is also a superset of Prolog [30]. Temporal operators in Tokio include: *concurrency*(,)(The clause "P:-Q, R" means that the Q and R are executed at the beginning of a time interval concurrently), *chop*(&&) (This operators divide a time interval into two subintervals), *next*(@), *always*(♯), *sometime*(<>), *keep*, *final*(fin) and so on.

Variables in a Tokio program may have different values at different time instances, i.e. the value of a variable varies with time. This makes the unification in Tokio more complicated. There are two kinds of unification in Tokio: one is concerned with unifying two Tokio variables, that is, unifying the entire sequences of values for the two variables. The second one is concerned with unifying the values of Tokio variables at specific moments in time through the use of special unification primitives.

Intervals can be manipulated using certain builtin operators, such as *length*, *empty*, and *notEmpty*.

The execution of a Tokio program is a mixture of resolution and transformation (or reduction).

The execution of a program in Tempura is a reduction or transformation process. Tempura is a temporal logic programming language in a broad sense, i.e. it is not based on the "logic programming" paradigm (resolution and unification). The execution of a program in Tokio is also a reduction process, but one which is combined with resolution and unification. The roots of Tempura are in the functional programming, imperative programming, and logic programming paradigm. There have been no attempts at developing either the declarative or the operational semantics of original Tokio programs. In order to give a formal semantics to Tokio, one would need to combine the semantics of ITL with a semantics of Prolog that explicitly represents the execution mechanism.

3.6.3 The Language

The standard operations on expressions in Tempura are available like $+$, $-$, $*$, $/$, div, mod, $=$, $>$, or, and . Tempura programs include eight elementary operators. As usual, "exp_i" means an arbitrary expression, b stands for a boolean expression and p and P' stand for programs. They are:

Equality: $exp_1 = exp_2$.

Parallel composition: $p \wedge p'$.

Conditional: if b then p else p'.

Local variables: $\exists v : p$.

Termination: $empty$.

Next: $next\ p$.

Sequential composition: $p; p'$.

Iteration: $p*$.

Variables in Tempura have the same syntax as in ITL. The basic statements (with the corresponding ITL constructs) are presented in Figure 3.1:

ITL	*Tempura*
$f_1 \wedge f_2$	f_1 *and* f_2
$A := exp$	$A := exp$
more	*more*
empty	empty
$\Diamond$	*sometimes*
$\Box$	*always*
true	*true*
false	*false*
if b then f_1 else f_2	if b then f_1 else f_2
while b do f	while b do f
repeat b until f	repeat b until f
"procedures"	*define* $p(e_1, \ldots, e_n) = f$
"functions"	*define* $g(e_1, \ldots, e_n) = e$

Figure 3.1: Basic Statements

A real Tempura program must be given the means to communicate with the outside world through *input* and *output*. Useful input and output facilities may need to be quite sophisticated. Two naive functions of *input* and *output* are given. The function *input* reads input from some device, such as a keyboard, and the function *output* produces output to another device, maybe a terminal. Both of these functions take a variable number of (zero or more) arguments whose values are read from the input device or written to the output device. Absence of *input* or *output* is denoted by a call of the corresponding function with no arguments: *input*() or *output*().

An example in Tempura The following is a very simple example to demonstrate a "while" loop. The user inputs the initial values of "M". The value of "M" in subsequent states is to be kept decremented by a constant value of 1. If the value of "M" equals 0, then the program will be terminated.

A sample Tempura program:

```
/* testwhile.t*/
/* Test run of While loop*/

define while_loop() =
 exists M,N: {
             input (M)
             and N=1 and
             always output (M)
             and
             while not (M=0) do
                 {skip
                  and M:=M-N}}.
```

The output:

```
Tempura 10> load "testwhile".

run while_loop().

[Reading file testwhile.t]

Tempura 11> State   0: > M=?

3.

State   0: M=3

State   1: M=2

State   2: M=1

State   3: M=0


Done!  Computation length:  3.  Total Passes:  4.

Total reductions:  72  (72 successful).

Maximum reduction depth:  9.




Tempura 12> load "testwhile".

run while_loop().

[Reading file testwhile.t]

Tempura 13> State   0: > M=?

0.

State   0: M=0
```

```
Done!  Computation length:  0.  Total Passes:  1.

Total reductions:  15  (15 successful).

Maximum reduction depth:  8.
```

3.7 Conclusion

Interval Temporal Logic, ITL, together with its executable subset of Tempura, offers a simple, expressive and efficient implementable framework.

Together with description of the assumption/commitment paradigm, which has been proved to be efficient to express time-critical systems compositionally, this chapter gave initial ideas and the formal basis (ITL and its framework) for the integrated approach developed in this research. An overview to ITL and Temporal Logic, Tempura and Tokio, and reasons of choosing ITL and its framework have also been given when describing our formal framework.

Chapter 4

An Integrated Framework

Objectives:

To define "Guided Evolution"

To outline the general framework

To describe the assertion points technique

To discuss visualisation and its advantage

4.1 Guided Evolution of Time-critical Systems

4.1.1 Why Guided Evolution?

千里之堤，溃于蚁穴。小之不慎，必成大祸。

This is a Chinese aphorism, lasting thousands of years. This ancient Chinese aphorism says: a great dam will collapse from a tiny hole dig by a wee ant (or a small leak will sink a great ship), and a catastrophic consequence will happen after ignoring a slender flaw.

This aphorism describes the key characteristic of time-critical, i.e., nothing is "tiny" or "small" in a time-critical system, so does any change. For example, we can imagine a time-critical system as a great dam, so a wrongly changed timing parameter (a wee ant) could cause a fatal failure of the entire time-critical system (collapse of the great dam). The evolution of time-critical systems is due to *changes* in the original requirements, adopting a different execution environment or to improve its efficiency. Continuous changes take place in a time-critical software life-cycle, even before the first release of a time-critical system. Unfortunately, time-critical systems behave like living things. Most time-critical systems never really become stable, although they can reach a so-called "steady state". For example, air traffic control systems need to be continuously evolved to cope with dramatically changing air traffics. A time-critical system may fall in a loop of evolution (Figure 4.1).

Due to the nature of time-critical systems, changes in them have a higher potential for producing impacts than other conventional systems. The tight coupling between functional properties and timing properties is the main cause of more impacts. A seemingly slight change (either functional or timing change) of requirements can bring massive new changes of both functional and timing properties. Furthermore, some timing properties won't prove to be satisfied until full implementation of the system. Therefore, the software-change cycle for time-critical systems can be much more complex. The key point is to manage the changes and their impacts with due regard to their increased complexity. Without the requisite change management, changes in time-critical systems can have unpredictable consequences.

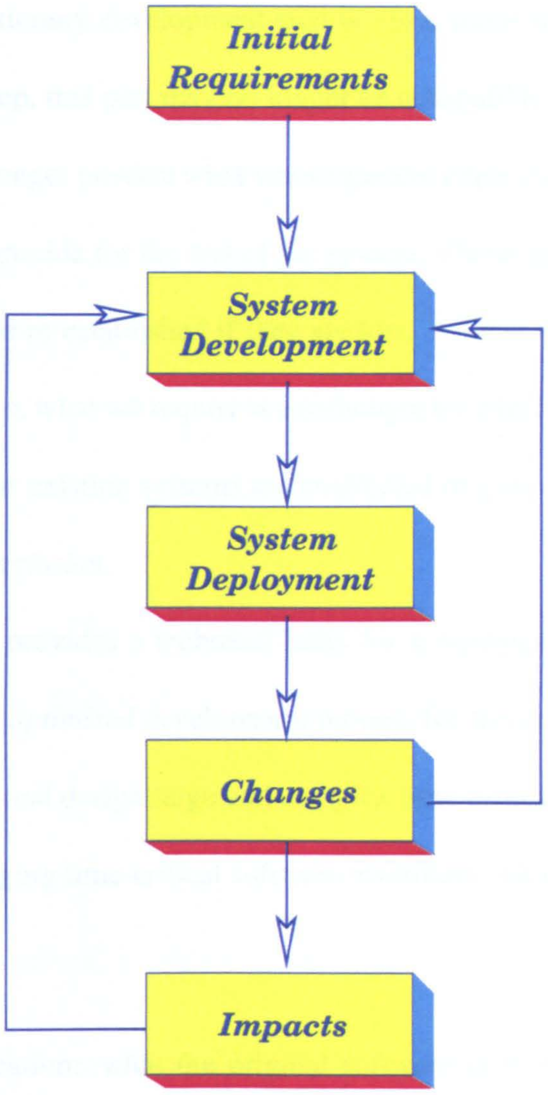


Figure 4.1: A Loop of Time-critical Systems Evolution

As we described in Section 1.1, we concentrate on certain aspects of change management, especially, managing impact of change with respect to identification, specification and control ripple effects of changes.

4.1.2 What is Guided Evolution?

During the evolutionary development, an important issue is to establish mechanisms to cope with propagation between different evolutionary steps, i.e. coping with *impact* of change. Every evolutionary development step is often made to a specific part of the system. After each step, this part may no longer be compatible with other parts of the system, as it may no longer provide what was originally expected or it may now require different services to provide for the rest of the system. These dependencies need to be checked, validated and re-established if they are lost. All this can be described as the *ripple effect*. Therefore, what we require is a technique by which evolutionary development is *allowed* so that existing systems are unaffected or gain a desirable effect. This is defined as *guided evolution*.

Guided evolution provides a technical basis for a *repeatable, well-defined, managed,* and potentially *optimised* development process for time-critical systems. It also shows how to specify and design large and complex time-critical systems. It addresses three aspects of managing time-critical software evolution. All these will be addressed in a formal manner.

(i) Formal Specification: what the original software is to do, what the proposed changes are to do, and what is the likely impact. In this work, the use of assumption/commitment framework can reveal or specify relationships between a system and its environment and among its sub-systems and can then specify impacts between different sub-systems and between the system and its environment. For example, for a sub-system, we can treat it as a self-contained system and treat its neighbouring sub-systems as its environment so that we can easily specify the impacts between this sub-system and its "environment" using assumption/commitment style specification.

(ii) Formal Design: how the new software is to accomplish its function including existing functions to be kept, newly required functions deriving from any changes, and functions required to meet impacts.

(iii) Refinement: the transition from requirements to implementation, i.e., the process of deriving code from formal specification through applying a set of mathematical rules (refinement rules), step by step, to guarantee the consistency between the formal specification and code.

It is essential to apply formal methods in time-critical systems evolution. A number of supporting reasons are as follows.

- Formal methods can specify properties of interest and increase our understanding of an evolving time-critical system and its changes by revealing inconsistencies, ambiguities and incompletenesses that might otherwise go undetected.

- The formal method, based on Interval Temporal Logic (ITL), used in this work is compositional. It is efficient to decompose a time-critical system, which usually is an extremely large system, into sub-systems with manageable size. This means that the correctness of the whole system should be verifiable in terms of the correctness of the decomposed sub-systems whose verification is computationally simpler. This is extremely useful when changes happen in a sub-system or a few sub-systems.

- Due to complexity and tight coupling between functional properties and timing properties, it is extremely difficult to specify a time-critical system or its changes by using conventional methods. Use of formal methods can cope this problem well.

- Indeed, we care more about performance than correctness for an evolved time-critical system. Therefore, use of Tempura, the executable subset of ITL, enables us to predict how well a system will operate in the field after applying changes.

The guided evolution technique is designed for the development of time-critical systems. The following are important characteristics of this technique.

- The technique centralises the formal specification of the system. The formal specification will be used or referred to throughout the whole evolution cycle. Furthermore, the technique covers both the time-critical system's formal specification and source code.
- It provides graphics (visualisation of time-critical systems), guidelines (mainly rules and some hints or tips of how to use rules), and a phase-based methodology.
- It ties program simulation, formal verification, source code implementation and testing in a unified manner.

In the rest of this chapter, a key point of the guided evolution, *assertion points* will be described in full. We will describe the phase-based methodology by splitting it into three phases. The core phase is the second phase, which uses two different abstraction levels, the *specification* level and the *source code* level. All phases are closely tied to the formal specification of the changes and the evolving system. Finally, a visualisation system will be depicted briefly. A set of formal guidelines will be given in Chapter 5.

4.2 Assertion Points

Introduction. We use an assertion points technique as a means of managing changes in time-critical systems.

Assertion points are at source-code level and divide a given code into pieces as introduced in Figure 4.3. For our purpose, the required information reflects the state of the system up to that point. A sketch of the analysis process is depicted in Figure 4.2.

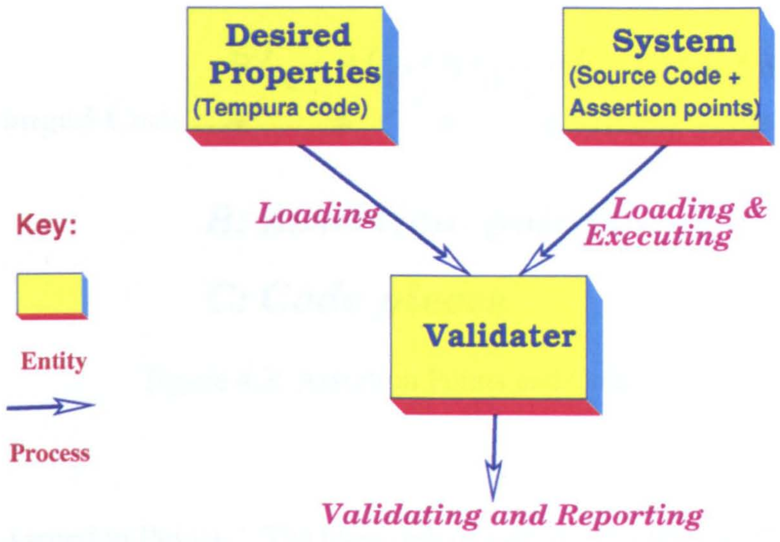


Figure 4.2: The Analysis Process

This figure gives a basic idea of the contribution in the thesis. Giving a property of interest we would like to check, we use assertion points in the code to check the validity of this property *at run-time*. For evolving time-critical systems, properties of interest (Desired Properties), which relate to changes, will be formulated and expressed in Tempura code. Assertion points will be added depending on different properties of interest. Assertion points technique consists of two components, assertion points to generate information and a mechanism to process this information. This mechanism will not only capture and interpret information generated by assertion points but also validate the properties. We give it an initial name of "Validator" here. The validator itself could be just a Tempura formula to represent the comparisons between run-time behaviours and properties of interest. We will give more implementation details in following chapters.

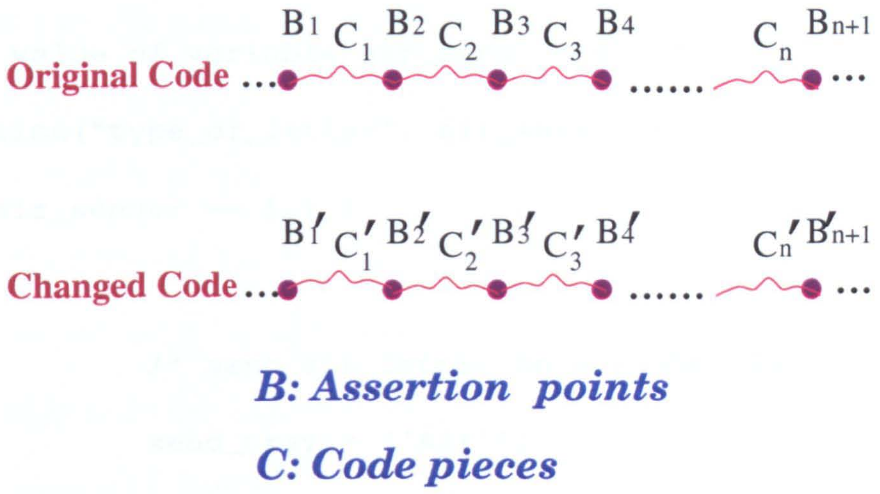

B1 C1 B2 C2 B3 C3 B4 Cn Bn+1
Original Code ... .
B1' C1' B2' C2' B3' C3' B4' Cn' Bn+1'
Changed Code ... .
B: Assertion points
C: Code pieces

Figure 4.3: Assertion Points and Code

The Task of Assertion Points. The main task of assertion points can be described as:

directly gathering and emitting *assertion data* from the "binary" level.

Assertion is a sequence of changes. Assertion data reflects run-time information, i.e., change of state.

The Construction of Assertion Points. The location of assertion points are chosen strategically. A simple strategy is to find out what variables used in expressing our property, to locate these variables in the source code, and put an assertion point directly after this location. We give a simple example, which is a part of a control program of a mail sorting system, processing air mail letter, as follows.

```
/* Detecting an Airmail letter and sending it
   to an Airmail tray */
air_sensor = 1 ;
/*an airmail sensor detects an airmail and sets
  the value of variable air_sensor to 1 */
assertion("type_of_letter", air_sensor);
 if (air_sensor == 1 ) {

             /* send air letter to a right tray */
             send_tray = ``Air'';
             assertion("tray_be_sent", send_tray);
               }
         }
```

where “air_sensor” indicates a result generated by an airmail sensor.“air_sensor = 1” indicates an airmail letter. “air_sensor = 0” means the letter is not an airmail letter. “send_tray” represents a control signal, which will be sent to an actuator to deliver the letter to an airmail tray. The two variables construct a property, i.e., whether the system can send an airmail letter to an airmail letter tray once an airmail letter has been detected. Therefore, two corresponding assertion points have been inserted directly after each assignment of a variable or related place, where the value of a variable will be changed, e.g., an assertion point, “assertion(“air”, 1)” , has been inserted just after the place, where the value of “air_sensor” is changed, to indicate whether the airmail sensor detects an airmail letter and send a correct signal or not.

Currently, the construction of assertion points has to be done manually. The construction process begins during step-wise refinement. As new variables are added, we begin to decide what assertion data will have to be produced and where assertion points shall be inserted. The real building and inserting process happens in the transformation process from low abstract level specification or Tempura code to executable code, such as C or Java code. The concrete assertion points are written in the same source code. Because the assertion points send sets of triplets, (variable, value, time stamp), an assertion point is a function or procedure (in conventional language, such as C) or a class (in object oriented (OO) language, such as Java), including three sub-functions/procedure (conventional language) or methods (OO language). The first captures the name of the variable, the second grabs the value of the variable, whose name has already been known by the first function/procedure/method, and the third function/procedure/method obtains the time stamp. An example, written in C style code, is presented as follows.

```
void aname()
{
    return getvarname();
}
```

```
void getval()
{
    return getvalue();
}

int myclock()
{
    return clock()/1000;
}

void assertion()
{
    printf("!PROG: assert %s:%d:%d:!\n",
                aname(), getval(), myclock());
}
```

Among this piece of code, the function, "aname", captures and sends a name of a variable, "getval" then reports the value of the variable, and meanwhile, "myclock" gives the time. Real Assertion Points are similar to this "pseudo-example", but could be more complicated. In addition, the function, *getvarname*(), is used to capture the name of the variable. *getvalue*() is a function to obtain the value of the variable. *clock*() is a C function, to return the processor time used by the program since the beginning of execution. $clock()/1000$ returns a time in millisecond.

Advantage of Using Assertion Points. There are a number of benefits of using assertion point techniques.

- Assertion points are easily built and used.
- Using assertion points is language independant.
- Assertion point technique lets the monitoring, analysing, and testing service of AnaTempura sit in-line without interrupting the system executing. Therefore, timing properties can be analysed and tested more succinctly.
- It offers an automatic, minimal and precise testing service.
- It helps the analysts to get to the root of the run-time problem, anywhere in the application, so analysts can quickly find the error or find new change impacts.
- It automatically pinpoints run-time errors quickly to ensure the reliability of the entire aspects of the property, not just part of it.

4.3 The Approach

In this section, a detailed description of our integrated approach will be given.

4.3.1 A Phase-based Methodology

A mechanism for managing change in a time-critical system should be *practical, systematic* and *compositional*. A fundamental issue in our approach is the ability to capture a possible behaviour of a (sub-)system. Once the behaviour is captured then we can assert if such behaviour satisfies a given property. And as a property captures a set of behaviours, *satisfaction* is achieved by checking if the captured system's behaviour is an element of this set. We are not dealing here with the formal verification of a property which requires that all possible system behaviours satisfy the property. However, the formal verification of the property may also be performed using an ITL verifier [95, 96, 112]. We are only concerned with validating properties which require only "interesting" behaviours to satisfy the properties.

As described before, the evolution of a software system could be due to "changes" in the original requirements, adopting a different execution environment or to improve its efficiency. The management of change is an important yet often problematic stage of the software development life-cycle. Even with substantial knowledge of a system, managing it's change and evolution is by no means straightforward. The introduction of a change to a single requirement may cause it to ripple through a system and impact on other requirements and broader organisational goals. The problem of identifying the total impact of a change is compounded by the size and complexity of relationships between requirements artifacts. This can make the process of assessing the effect of change expensive, time consuming and error-prone [78]. An essential way of managing change is to start with handling the changes at requirements level. To begin with, we must consider the relationship between the old and new requirements. Our approach for managing changes is an integrated process that considers impacts at both requirements and source code levels. The basic process is depicted in Figure 4.4. Its three basic phases are described below.

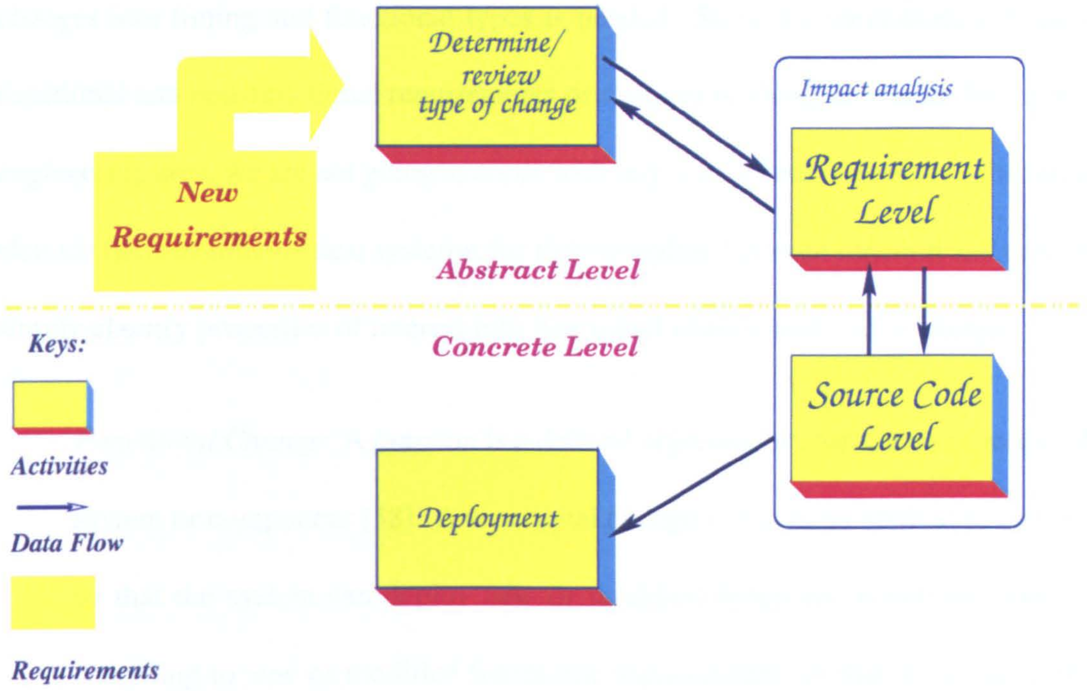


Figure 4.4: Impact Change Management Framework

Phase 1: Determine type of change

Phase Overview This phase reviews the proposed changes and checks the feasibility of changes. The aim of this phase is to identify the type of change.

Phase Activities Although the term, "review of new requirements", has not been put into the box of Phase 1, "Determine Type of Change" (This is due to limited place in the figure), essentially, we need a review of the new requirements before we can process the identification of the type of change.

The developer should make sure that he understands the change being proposed and the relevant system components affected, i.e. one needs to evaluate and clarify the change request. In time critical systems, a review and classification of requirement changes into timing and functional types is needed. Since the identification between functional and non-functional requirements or changes is a long debate in the software engineering area, we are not going to argue with any school here. Because of the unique characteristic of time-critical systems, the tight coupling between function and time, we simply classify properties of interest into functional change and timing change:

Functional Change: A *function* is a defined objective or characteristic action of a system or component [58]. A functional change is a change applied to a system so that the system can deploy new or modified functions, which are specified according to new or modified functional requirements. In this thesis, we define functional change as all non-timing changes.

Timing Change: Timing changes are applied to modify existing timing properties, i.e., changes of values of timing parameters and then timing behaviours.

One can say that timing change could be a part of functional change. There are intersections between functional and timing changes. For clarifying the problem in the thesis, we classify all inter-sectional changes about timing parameters into timing change category. Then considering timing and functional requirements as two entities in system development, we can describe the changes in either of them as a co-evolution process, i.e. some functional requirements bring new timing requirements or some timing requirements bring new functional requirements. It is proposed therefore to study the effect of changes of one entity on the other. This is in addition to the relation between their rates of evolution and their natural interaction. This plays a crucial role in fixing the shape and character of the changes, as well as the kinds of solution we can have, and the types of method we can use to obtain them.

Phase Outcomes This stage will result in the identification of the current system requirements specification before changes plus a review of the new requirements and their allocations in the current system. The outcome of this stage will be a description of the change and the type of the change in text. If the change brings modification of the number of timing parameters or the value of any timing parameters, then the change affects some timing properties of the system.

Phase 2.1 Impact analysis: Requirements level

Phase Overview At this stage, all timing and functional requirements will be analysed and the effect of the required change will be determined. Strategies will be devised to minimise the effect of changes. Formal specifications of properties of interest should be made in this phase.

Phase Activities The major activity of this phase is impact analysis at a higher abstraction level. As we mentioned before, one identifies the potential impact of the changes on the whole system, i.e., hardware/software system, documentation, people, etc. Impact analysis can also aid in the communication of the complexity of requested change to the customers or end users who proposed the change. The results of impact analysis can be used for later analysis, mainly, *dependency analysis* [17, 102] and *traceability analysis* [17].

- *Dependency*: the ability to examine and evaluate dependency relationships among program entities, such as data, control and component dependencies. *Dependency analysis* is to track relationships between different sub-systems and to analyse impact of any change.

- *Traceability*: the ability to trace a design representation or actual program component back to requirements. *Traceability analysis* is to trace system requirements through design, code and testing. Our run-time analysis is a powerful way to perform this kind of analysis by linking design (specification), code and testing altogether.

The main technique we used in this phase is based on the assumption/commitment framework and compositional technique. This two techniques will be used to reveal and specify changes and impact of changes. We use the assumption/commitment framework to specify each change or property of interest separately. That is, when specifying a change and a related target sub-system, we treat all surrounding sub-systems as the target sub-system's environment. Let's recall the assumption/commitment paradigm (Section 3.4. The form of assumption/commitment specification in ITL is as follows:

$$w \wedge As \wedge Sys \supset Co \wedge fin\ w',$$

Giving a system, we would like to use this formula to distinguish between a target sub-system and its environments. The environment of a target sub-system could be its surrounding sub-systems or possible external environments, i.e., external factors of the whole system, with which the target sub-system communicates directly. The sub-system, Sys, starts in a state, which satisfies w. For it to terminate in a state, satisfying w', and committing to a property described by Co, the sub-system's environment must achieve properties expressed by As, which presents assumptions or restrictions placed over the target sub-system's environments. If we apply a change to a target sub-system and a new assumption should be place against a certain neighbouring sub-system, the neighbouring sub-system should be changed correspondingly. Therefore, the new change, required to this neighbouring sub-system, is caused by impact of change to the target sub-system. Through this way, we can then specify the impact of change.

Phase Outcomes The result of this stage is a formalisation of all properties of interest. These requirements are expressed as ITL formulae. The soundness of the requirement specifications can be checked by a tool support environment, which includes an interactive theorem prover [95, 96] based on the PVS [112]. The benefit for both end users and software designer is that changes with unrealistically large impacts will be prevented from being implemented. However, specification checking is not a part of the thesis, we can use work done in [95, 96] and [112], if we want to check the soundness of the required specifications.

Phase 2.2 Impact analysis: Source Code Level

Phase Overview This phase involves the implementation of the proposed changes and some further, necessary impact analysis in conjunction with each developing step. The development from initial specifications in ITL, obtained from Phase 2.1 (Section 4.3.1), is achieved through step-wise refinement using our ITL refinement calculus. During the development process, assertion points will also be inserted into source code. Once we derive source code from the specifications, then we can perform the main task of this phase, run-time analysis and validation. The formal specification will be referred throughout this phase.

Phase Activities This phase concentrates on the impact analysis at source code level with respect to formal specifications. This analysis will be accomplished at run-time. The run-time analysis checks whether an implementation is correct with respect to a corresponding specification. It checks the consistency between the domain of code and the domain of formal specification. It also detects further change impacts which only appear during or after the implementation. In most cases, such impacts refers to timing properties. The run-time analysis will be performed using AnaTempura via Assertion points (see section 4.2).

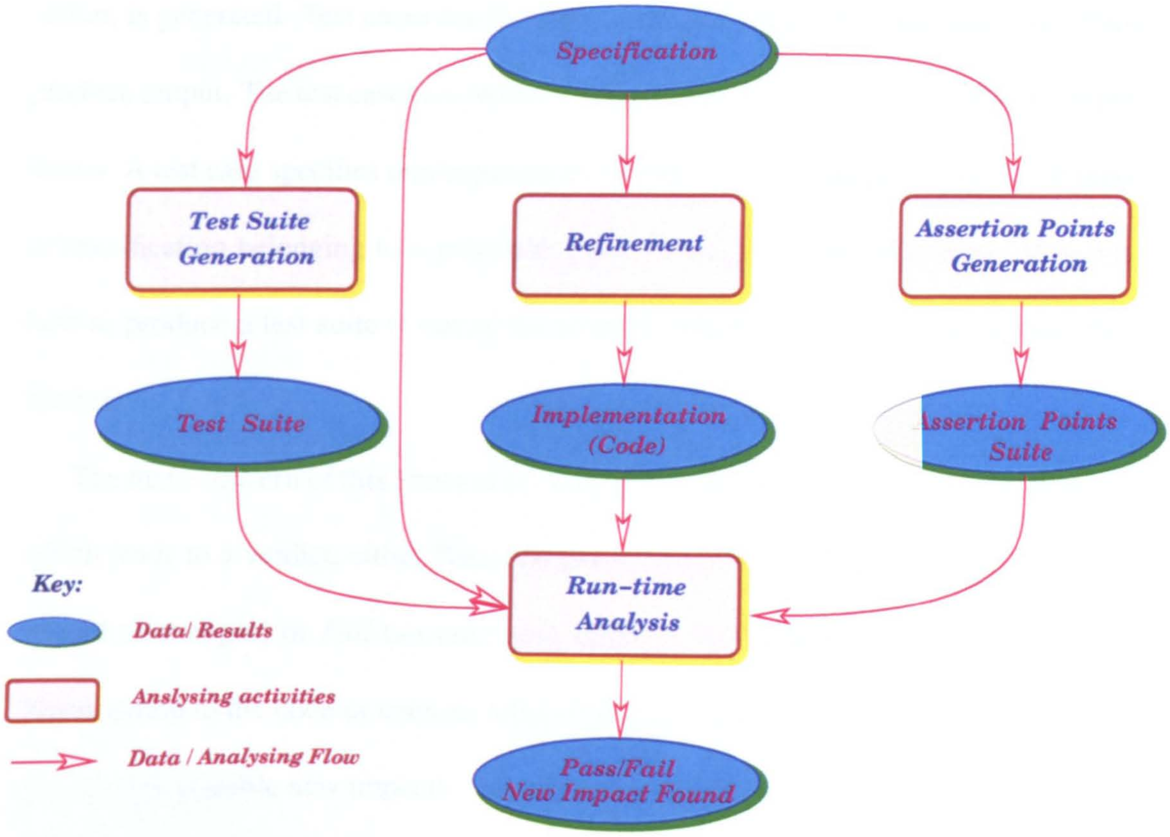


Figure 4.5: Impact Analysis at Code Level

In Figure 4.5, the impact analysis at source code level, the run-time analysis, is depicted. Starting with a formal specification, obtained from Phase 2.1, an implementation (code) has been developed via step-wise refinement and programming; it is, for example, a C program. By means of *run-time analysis*, we would like to check whether the final system satisfies the specification and whether a new impact is found after implementation. To this extent, an *assertion points* suite is generated in conjunction with implementation (code). The assertion points suite is embodied in the source code. This suite is used to realise the *run-time analysis*. More details of the assertion points will be described in next section.

If necessary, an abstract *test suite*, i.e., a set of test cases derived from the specification, is generated. Test cases are the input data values for which the property should produce output. The test cases are in the domain of the function that the software implements. A test case specifies one experiment related to one test purpose and to one piece of specification belonging to a particular part or component of the system. However, how to produce a test suite is out of the range in this thesis and will not be discussed further.

The main concern of this phase is the subsequent execution of the *run-time analysis*, which leads to a verdict, either *Pass* (the implementation conforms to its specification related to changes) or *Fail* (an error was found). The verdict of "Fail" results in an investigation to the code or even recalling Phase 2.1 to find out what causes the failure and finding possible new impacts. Results are generated automatically by the run-time analysis via a monitoring process of information generated by assertion points. More details of the analysis process is set out in the next chapter.

Phase Outcomes Because this phase involves refinement and programming, the first outcome is the source code with assertion points inserted. The next outcome is results generated from run-time analysis, including a record of the run-time analysis and possible animation of run-time behaviours. The record of the run-time analysis includes two possible files. One file is a log file, which records whether it is *Pass* or *Fail* for each comparison between run-time behaviours and properties of interest. If the result is *Fail*, two different values and their location will also be recorded. In this case, new impact of change might be found. Another file contains all values of timing parameters produced during the run-time analysis.

Phase 3 Deployment

Phase Overview We will review all files or records produced during the previous phases. We will then generate documentation of the system. A manual will be produced if required. The whole system, together with the updated documentation, formal specification and evolution records, should be ready to be deployed after this phase.

Phase Activities The main activity of this phase is to generate other system deliverables than the system itself, including an evolving or developing history, documentation and a possible manual. The document will be a combination of diagrams, graphics and text, together with the ready system.

An objective of reviewing process in this phase is regarded as a formal operation to capture and record specific aspects of the process of handling changes in time-critical systems evolution. The purpose is to rigorously review and document the following:

- What was wanted in the first place from the change requirements.
- What the change promised to do.
- What the change actually did.
- What was the change impact.
- Design and performance features of the applied changes and the finished system.

- The whole history of the modification.

One of the first documents to be generated is a preliminary statement of change requirements. This basically defines the purpose and intended new properties of the proposed system. This document will be produced after a review of outcomes from phase 1.

The most important document is a number of formal specifications in ITL and Tempura, which express all properties of interest, as well as impact of changes found in phase 2.1. We will collect all related formal specifications and write them into a separate file. A set of tables, diagrams, or figures will also be included in this document if we have any, for example, a table containing all timing information.

The next document will be a collection of log files produced from phase 2.2. A minimum requirement of the document is to include a log file generated from the run-time analysis. Other documents can be a trace file that records timing information for all states, a set of screen shots produced during animation, and a log file produced by AnaTempura that can include information about the run of the system, for example, test cases injected by AnaTempura or any error message echoed by the system. In some cases, we can generate a file containing assertion data.

Another possible activity is to add some necessary comments into the source code body, for example, marking newly added code pieces or commenting changed code pieces.

A manual will be produced or updated in this phase if required. The manual gives necessary instructions of how to use the system, features of new functions, difference between old and new systems, instructions of how to use new functions, and some possible troubleshooting.

Phase Outcomes Documentation of the system will generated in this phase. A checking list of outcomes includes:

- a description to change requirements,
- related formal specifications of changes and related sub-system,
- the source code itself with a set of assertion points inserted,
- a log file, containing all information of run-time analysis, generated by AnaTempura, as well as necessary explanations,
- a trace file, containing timing information, i.e., values of timing parameters in different states,
- a set of screen shots captured from animation,
- a log file recording all messages produced by the system during its run,
- a set of necessary comments added to the source code after the run-time analysis, and
- a manual if required.

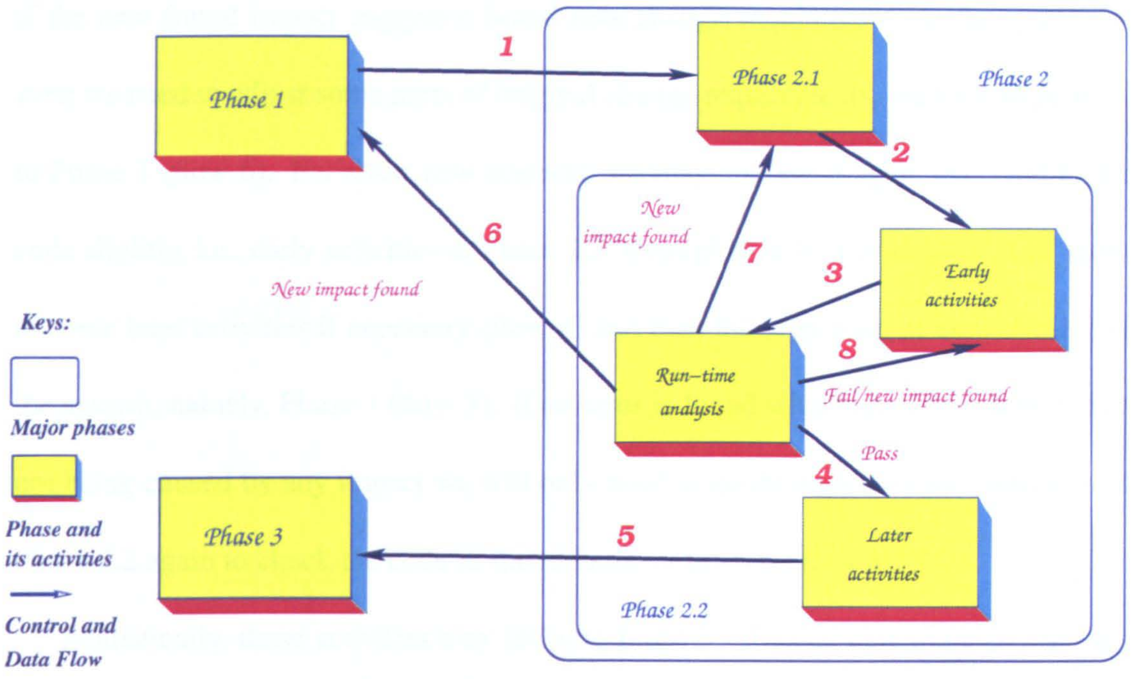


Figure 4.6: The Role of Run-time Analysis

Relationships among Phases

As we described before, *run-time analysis* of the Phase 2.2 plays a pivotal role in the approach. As we can see in Figure 4.6, the most number of flows (flow 3, 4, 6, 7, and 8) link to run-time analysis. Results obtained from run-time analysis decides whether we can turn to the next phase or recall the previous phases. This is depicted in Figure 4.6.

In Figure 4.6, "Early activities" include various phase (2.2) activities before run-time analysis, such as refinement, and insertion of assertion point. "Later activities" include various phase (2.2) activities after run-time analysis, such as a draw of timing diagrams. "Later activities" only happened when all tests are passed during run-time analysis. If any problem happens during run-time analysis, that indicates either failures of the source code or new impact. Decisions will then be made. At any time, if new impacts have been found, we need to go back to Phase 2.1 through the flow 7. Sometimes if the new found impact triggers a brand new change requirement for the system or even we need to adjust some parts of original change requirements, we have to go back to Phase 1 (flow 6). For some new impacts, we may analyse it again and modify the code slightly, i.e., early activities of Phase 2.2, through flow 8. The verdict, pass, leads to some later activities if necessary (flow 4) and then the beginning of employment of the system, namely, Phase 3 (flow 5). If an error is found which has been identified as not being caused by any impact we will only need to go through the early activities of Phase 2.2 again to check the code or the refinement process (flow 8).

Realistically, these activities may fall into loops if errors or new impacts keep appearing. Loops could be from flow 3 to flow 8, which indicates inconsistency between specifications and the source code. A loop with flow 2, flow 3, and flow 7 or a loop with flow 1, flow 2, flow 3 and flow 6 means continuously appeared new impact. An ideal path will consist of flow 1, flow 2, flow 3, flow 4 and flow 5. If a sub-system is small or simple enough, the ideal path will likely happen. Therefore, in this research, we try to use compositional and incremental manners to reduce loops, especially loops, consisting of flow 1, flow 2, flow 3 and flow 6. As a solution, we absorb the infrastructure of the Incremental Model [84], replacing its elements of the linear sequential model by the general structure of our phase-based methodology. Furthermore, under the compositional manner, instead of producing an increment each time, we compose a certain number of related increments. This produces a modified incremental model (Figure 4.7). We shall notice that new increments, such as, "Increment 1+2" or "Increment 3+4", may still need some run-time analysis in some rare cases. Because of the benefit of using compositional rules and introducing some necessary strong assumptions, the soundness of new increments should be guaranteed in most cases without further run-time analysis or testing.

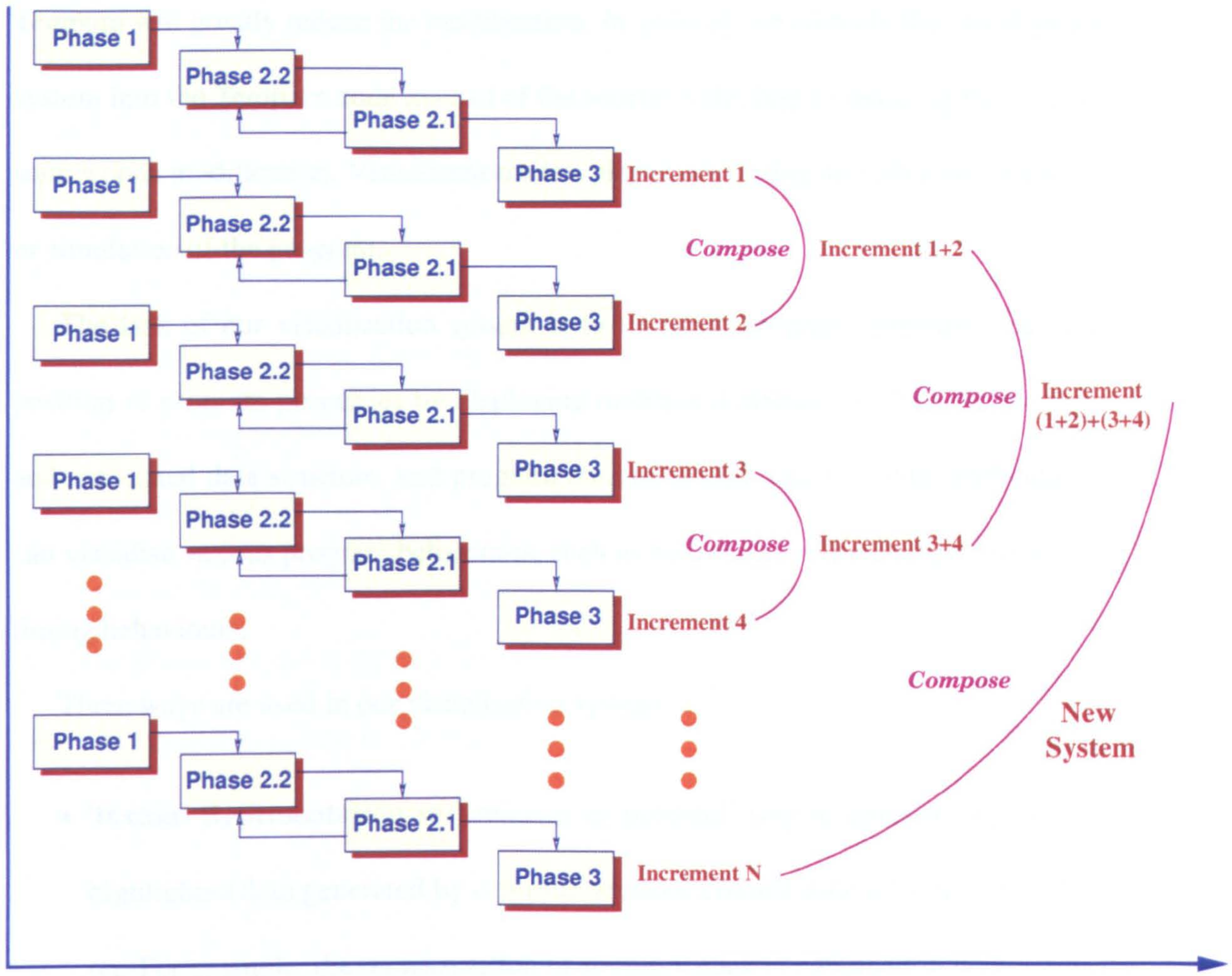


Figure 4.7: The Modified Incremental Model

4.3.2 Visualisation of Time-Critical Systems

A visualisation system has also been employed in our methodology. This visualisation system can help developers in selecting and specifying pictorial representations which can easily be understood, during a time-critical system's evolution.

Visualisation, in essence, forces the developers to modify the program (the source code), as well as related parts of its specification. In our visualisation system, the use of Tempura will greatly reduce the modification. In general, we embody the visualisation system into the Tempura code instead of the source code, highly reducing the level of source code modification. Visualisation takes place both during and after the execution or simulation of the program.

The task of our visualisation system is to visualise program behaviour, i.e., exposition of program properties by displaying multiple dynamic views of the program and associated data structure, and program animation and data-structure rendering. It can visualise various program behaviours, such as behaviours of parallel programs and timing behaviours.

Three ways are used in our visualisation system.

- **Textual Representation** still remains as potential way to dynamically display highlighted data generated by run-time process, control flow information and so on. For example, the representation of timing values in conjunction with animation images in the animation screen.

- **Timing Diagrams** present program data in a two-dimensional way with time information on one axis and variable values on the other.

- **Animation** of the program execution symbolically places each process, sometimes with a portion of distributed data, on an individual point in a two dimensional display, corresponding to a single instant of time, i.e. a snapshot. As time advances, the display changes like an animated movie.

The animation system defines a number of objects - usually considered to be animation images (icons) - with parameters. These objects can be changed by program operation. The object parameters are bound to program variables which are generated by assertion points. These assertion points interact with a software monitor system, which can listen to/ record all messages during execution and pass this information on to the visualisation system. Multiple graphic views of properties will be animated by highlighting an operation or a process with blinking, changing colours, and moving animation images on the screen.

In our visualisation system, the animation images have been designed to loosely correspond to the program's data or agents, i.e. the animation images represent abstractions designed to elucidate the program semantics. The visualisation system provides a pictorial representation at a proper abstraction level, as well as the Tempura code of the system. This representation is enough for developers or analysts to understand what the system do, how it works in either a correct way or incorrect way.

Our visualisation system shows timing behaviours via a time diagram, which indicates the state of processes. The time diagram visualisation works as follows:

1. Timing values (time stamps) for each process are first generated by assertion points and sent to a trace file generator in the same order as during run-time.

2. A *trace file* will be produced after the program execution.

3. An animator of the visualisation system will read the *trace file* and then represent it in a pictorial manner.

Time diagram visualisation takes places after the program execution. Accurate simulations of real-time behaviours will take place during the execution of the program. This will be realised by sets of assertion points. In this case, these assertion points send information to the visualisation system at each breakpoint in run-time. Then the visualisation system displays the time information in a textual manner on the animation screen, corresponding to each animation image.

The employment of the visualisation system will enable developers or users to:

- observe how computations are executed,
- visually develop and analyse algorithms and system programming
- analyse and improve the interactive run time facilities,
- monitor and analyse performance of the system,
- assist the users in choosing implementations, and
- produce documentation and description of algorithms and time-critical programs.

In addition, this system supports different programming languages, like C/C++, Java, and Verilog.

4.4 Summary

In this chapter, we presented a general methodology for handling change during time-critical system evolution. Using the technique, we can validate and analyse those behaviours of a system which are of interest.

The methodology provides analysis at different abstract levels that automatically guide the users to the location of program bottlenecks and change impacts. It also provides ways to observe how computations are executed. It provides a way of development and analysis of change and system programming, monitoring of performance and other aspects, as well as documentation and description of new design and programs.

The visualisation facility generates alternative views in a graphical manner throughout the whole process. It acts as comprehension aids to formal techniques applied in the methodology.

Chapter 5

Evolution: Guidelines and Analysis

Objectives:

To describe run-time analysis in a formal way

To discuss compositionality for formal evolution

To discuss timing analysis for formal evolution

5.1 Defining Run-time Analysis

A key task of our approach is to check a time-critical system at run-time and analyse whether the (run-time) behaviours of the system satisfy its properties of interest. For building such a execution mechanism, we need to specify the run-time analysis processes. A simple ITL formula is used to define the run-time analysis as follows:

$$Sys \ \wedge \ Prop \ \wedge \ Monitor$$

where Sys denotes the real system (code)'s run-time behaviours, $Prop$ denotes those properties that we want to check, and $Monitor$ represents the checking process. This formula is not as simple as it seems. We give more details as follows:

Sys reads in the sequence of changes, as generated by the system via the assertion points. Furthermore, it converts this sequence into a behaviour (interval).

$Prop$ stands for properties of interest we are going to check. It is actually a formal specification, written in Tempura. It tells what the system should do. The Tempura interpreter will generate the behaviour corresponding to $Prop$.

$Monitor$ will compare the behaviour of Sys with that of $Prop$.

Now we describe Sys in more detail. In fact, because of the use of assertion points, Sys can be simplified and specified as follows:

$$\Box\ (input\ Sys\_AD)$$

where Sys_AD means assertion data of a running external program/system that is a list of variables and their instant values. This is a simplified version of Sys. In fact, once Sys reads a sequence of changes, it will convert the sequence to a behaviour.

5.2 Compositionality

When hardware designs are considered, a typical solution is to partition the whole system into units/modules and handle each unit/module separately. This is called the modular approach. Similarly, this solution can be applied to software design. Large software systems with unmanageable size and complexity can be decomposed into smaller and manageable sub-programs or components. These sub-programs or components will be treated in separation, based on an assumption about the behaviour of its surrounding sub-programs, components and even the environment of the whole system. In this case, these surrounding sub-programs and components can be said to be the environment of the sub-program or component with which they are being handled. This section describes the notation of *compositionality* with aspects of real time critical systems, i.e. firm links to timing analysis guidelines, and compositional theories, developed for handling time-critical systems' evolution.

5.2.1 The Role of Compositionality

Compositionality is an important property for development and verification of any sizeable system [34]. It is also a crucial technique to guided evolution. Due to the very complex nature of time-critical systems, it is extremely difficult to evolve such a system. As described in the previous sections, one ideal way is to develop, verify and test components or subsystems separately, to integrate components and subsystems and to commission the whole system. There are two reasons to support this method:

- The full development is very complicated and takes a long time to accomplish. It is therefore in general very difficult to verify the system after the whole construction, because all the useful intuition used during the design process will be lost and the final verification can be as hard as building the whole system from scratch again.

- If the verification is only conducted after the implementation, early design errors will result in loss of most of the later design work. The engineers may need to return to the very beginning again.

Compositional methods allow gradual development, which enables the satisfaction of a specification by a system be verified on the basis of specifications of its constituent components, without knowing the interior construction of these components. It also allows a so-called *verify-while-develop* paradigm [132].

In 1981, Jones [67] proposed a way to describe the interference between a component and its environment by a *rely*-condition and a *guarantee*-condition. These represent state changes from the environment and the component respectively. In this way, compositionality can indeed be achieved. Jones also gave an outline of a compositional proof system for a large subset of the Owicki and Gries language [67]. An initial attempt to give a semantic model for Jones' system was made by Aczel (see [34]). The first mathematical treatment of this approach was developed by Stirling [119]. Further work includes [120, 123, 122, 121, 134, 131].

5.2.2 Compositional Method

Compositionality asserts that *the specification of a program should be verifiable in terms of the specification of its syntactic subprograms* [34].

As described in Section 3.4, the assumption/commitment paradigm will support compositional evolution and methods. The assumption/commitment technique is valuable as a compositional principle to be used during guided evolution. ITL has a compositional proof system and has refinement rules. It also offers a workbench for writing specifications and for verifying them compositionally. An attempt at developing compositional theories based on ITL workbench and a synthesis of existing work has formed part of this research. In detail, our work is based on Moszkowski's [95] and [96]. We attempt to extend ITL workbench by adding assumption/commitment style specifications and compositional rules with adopting Cau's paradigm, [25]. This supports the approach described in Chapter 4 and plays a key role in the whole integrated framework of the evolutionary development of time-critical systems.

A composition principle gives a way of composing assumption/commitment specifications while discharging their assumptions [89, 104, 11, 2].

We recall the format of assumption/commitment specifications, which has been described briefly in Section 3.4. A system uses certain conditions (assumptions), w.r.t. the system's environment, and gives expected behaviours (commitments) under these conditions. Such a system is specified as follows:

$$w \wedge As \wedge Sys \supset Co \wedge \mathit{fin}\ w',$$

The system consists of a quadruple, (w, As, Co, w'). Here w and w' are state formulas whereas As, Sys, and Co are arbitrary formulas which can contain temporal constructs. The implication says that if w is true in the first state and As is true for the period when Sys is active, then Co also holds and that w' is true in the final state. A system Sys satisfies such a specification, denoted by the formula

$$R_{Sys}(w, As, Co, w'),$$

if Sys is invoked in a state which satisfies w, and any environment action satisfies As, then any component action satisfies Co, and if a computation terminates, the final state satisfies *fin* w'. Valid computations are those that if they satisfy assumptions, then they also satisfy commitments. A program satisfies a specification if all its computations are valid.

For example, informally, in a nuclear power station control system, a property of interest is described as:

> Nuclear reactors of a nuclear power station will eventually be shut down once the level of the radioactivity is exceeding the safety alert level.

An assumption/commitment specification of the property, P, is given by:

$$w \wedge As \wedge P \supset Co \wedge \mathit{fin}\ w'$$

$$\begin{array}{ll}
w: & \mathit{Reactors\_run} = 1 \wedge \mathit{Em\_Stop\_Send} = 0 \\
As: & \mathit{true} \\
Co: & ((\mathit{Radio\_level} \geqslant \mathit{Alert}) \supset (\mathit{Em\_Stop\_Send} = 1 \wedge \bigcirc \mathit{Reactors\_run} = 0)) \vee \\
 & ((\mathit{Radio\_level} < \mathit{Alert}) \supset (\mathit{Em\_Stop\_Send} = 0 \wedge \bigcirc \mathit{Reactors\_run} = 1)) \\
\mathit{fin}\ w': & (\mathit{Reactors\_run} = 0 \wedge \mathit{Em\_Stop\_Send} = 1) \vee \\
 & (\mathit{Reactors\_run} = 1 \wedge \mathit{Em\_Stop\_Send} = 0)
\end{array}$$

where variables, Em_Stop_Send and $Reactors\_run$, are boolean variables. The value of $Reactors\_run$ equals to 1 and Em_Stop_Send equals to 0 in the initial state. If the value of $Reactors\_run$ equals to 1, then nuclear reactors keep running. If the value of $Reactors\_run$ equals to 0, then nuclear reactors terminate. The value of $Reactors\_run$ does not change until the value of Em_Stop_Send equals to 0. The value of Em_Stop_Send is decided by $Radio\_level$, which indicates the level of the radioactivity. If the value of $Radio\_level$ exceeds the threshold, $Alert$, i.e. the level of the radioactivity is beyond the safety level, the control system sets the value of Em_Stop_Send to 1 and then the reactors stop. Otherwise, the initial state will be kept, i.e., the reactors run.

Refinement Further, refinement may be expressed as:

$$((As \wedge w) \supset (Co \wedge \mathit{fin}\ w')) \sqsubseteq Sys$$

"$\sqsubseteq$" means refinement. It can be abbreviated to

$$(Ass \supset Com) \sqsubseteq Sys$$

where $(As \wedge w)$ is represented by Ass and $(Co \wedge \mathit{fin}\ w')$ is abbreviated by Com.

The main rule for guided evolution is then as follows:

Let Sys be the system that has to evolve, X be the 'addition' and $C(Sys, X)$ be the evolved system.

$$\frac{\begin{array}{l}(w \wedge As) \supset (Co \wedge \mathit{fin}\ w') \sqsubseteq Sys,\\ (w_x \wedge As_x) \supset (Co_x \wedge \mathit{fin}\ w'_x) \sqsubseteq X,\\ R_C(w_c, As_c, Co_c, w'_c, w, As, Co, w', w_x, As_x, Co_x, w'_x)\end{array}}{(w_c \wedge As_c) \supset (Co_c \wedge \mathit{fin}\ w'_c) \sqsubseteq C(Sys, X),}$$

where C is a system composition operator such as sequential composition (;), conditional, iteration, parallel composition, etc. The condition

$$R_C(w_c, As_c, Co_c, w'_c, w, As, Co, w', w_x, As_x, Co_x, w'_x)$$

is the condition under which the ‘adding’ of X to Sys is sound. Furthermore, in the evolutionary development process, X could be a sub-system, which is awaiting to be composed into the whole system. In this case, Sys will be the previous built sub-system whilst $C(Sys, X)$ indicates the final system. The compositional rule aims at proving the correctness of $C(C(Sys, X)Sys, X)$, i.e. the composition of Sys and X, from the correctness of Sys, the correctness of X and relations between the corresponding assumption/commitment specifications.

In addition, if we abbreviate $(As \wedge w)$ to Ass and $(Co \wedge \mathit{fin}\ w')$ to Com, then the general rule can be expressed as follows:

$$\boxed{\dfrac{\begin{array}{l}(Ass \supset Com) \sqsubseteq Sys, \\ (Ass_X \supset Com_X) \sqsubseteq X, \\ R_C(Ass_C, Com_C, Ass, Com, Ass_X, Com_X)\end{array}}{(Ass_C \supset Com_C) \sqsubseteq C(Sys, X)}}$$

where the condition

$$R_C(Ass_C, Com_C, Ass, Com, Ass_X, Com_X)$$

is the condition under which the ‘adding’ of X to Sys is sound. The derivation of the final system is performed using our refinement calculus. The calculus applies a method in which the system is constructed gradually, step by step, using sound refinement rules, to its final implementation. Each new design step for a new change is checked as soon as it is carried out, and the correctness of these individual steps together guarantees the correctness of the whole, new system. This calculus is a formal guideline to our general approach, presented in Chapter 4, the step-by-step methodology (Figure 4.4), and modified incremental model, presented in Figure 4.7.

5.2.3 Sequential Composition

If C is sequential composition then the rule is as follows:

$$\boxed{\dfrac{(Ass \supset Com) \sqsubseteq Sys, \quad (Ass_X \supset Com_X) \sqsubseteq X, \quad Ass_C \supset Ass, \quad Com \supset Ass_X, \quad Com_X \supset Com_C}{(Ass_C \supset Com_C) \sqsubseteq (Sys\,;X)}}$$

where X is another system, which will be added to Sys, sequentially. Ass and Com represent the assumption and commitment of Sys, respectively. Ass_X and Com_X represent the assumption and commitment of X, respectively.

The rule is a derivable proof rule. It tells that we can derive a new system, $Sys\,;X$, from the sequential composition of an existing system, Sys, and another sub-system, X, if and only if all premises of the rule hold. Among these conditions, $Ass_C \supset Ass$, $Com \supset Ass_X$, and $Com_X \supset Com_C$, must hold to guarantee the correctness of the final composed system, $Sys\,;X$, with respect to Ass_C and Com_C.

The condition, $Ass_C \supset Ass$, denotes that the assumption of the composed system, $Sys;X$, must imply the assumption of the existing system, Sys, i.e., the environment of the new system, $Sys\,;X$, should be at least as strong as the environment of the existing system, Sys. The condition, $Com \supset Ass_X$, indicates that the assumption of the sub-system, X, must be implied by the commitment of the existing system, Sys, i.e., the environment of X should be at least as strong as the commitment of the new system. The last condition, $Com_X \supset Com_C$, specifies that the commitment of X must imply the commitment of the new system, i.e., the commitment of the final system must be at least as strong as the commitment of the sub-system, X.

Demonstrative Example

Recall the example given in Section 3.4, we reuse the specification of $PumpController$ and present it again as follows:

$$
\begin{array}{l}
(Ass, Com), \quad where \\
Ass : true \\
Com : \Box \; (\\
(Pressure \geqslant MineExplo \supset \Diamond(Pump = OFF)) \wedge \\
(Pressure < MineExplo \supset \Diamond(Pump = ON)))
\end{array}
$$

Supposing, in the real system, if the pump is switched off, an alarm should be activated. We define the sub-system for controlling the alarm, X, as follows:

$(Ass_X, Com_X), \quad where$

$Ass_X : true$

$Com_X : \Box\,($

$(Pump = OFF \supset \Diamond(Alarm = ON)) \wedge$

$(Pump = ON \supset \Diamond(Alarm = OFF)))$

We then add Sys and X together, using the above sequential composition rule. The new system, C, is defined as follows:

$(Ass_C, Com_C), \quad where$

$Ass_C : true$

$Com_C : \Box\,($

$((Pressure \geqslant MineExplo \supset \Diamond(Alarm = ON)$

$\wedge$

$((Pressure < MineExplo \supset \Diamond(Alarm = OFF)))$

Apparently, the premises of the rule hold, i.e., the rule can be applied.

5.2.4 Parallel Composition

Most time-critical systems are inherently parallel. The correct behaviour of a parallel program is critically dependent on synchronisation and communication between processes. *Synchronisation* is the satisfaction of constraints on the interleaving of the actions of different processes. *Communication* is the passing of information from one process to another. Synchronisation and communication are linked to each other, because some forms of communication require synchronisation, and synchronisation can be considered as contentless communication. In parallel programming, communication between processes is modelled in two ways, *shared variables* and *message passing*. Shared variables and message passing are used to express state-based parallel processes and message-based processes respectively.

Shared Variables are objects that more than one process have access to; communication can therefore proceed by each process referencing these variables when appropriate [23].

Message Passing involves the explicit exchange of data between two processes by means of a message that passes from one processes to another via some agency [23].

In the thesis, we concentrate on state-based parallelism because the formal base of our approach, ITL, is state-based and to develop a full set of composition rules is out of the range of the thesis. Further, parallel rules have been devised to compose assumption/commitment specifications of parallel processes. These composition rules are usually hard to construct because of mutual dependency: each process contributes to the environment of the other ones and the commitment of a process thus influences the assumptions of the other ones. Although this problem exists whatever communication model is adopted, the corresponding assumption/commitment methods evolved into different rules for parallel composition [25]. In the state-based approach, a typical premise of the rule for deducing a specification of $Sys_1 \parallel Sys_2$ from the specifications of Sys_1 and Sys_2 is of the form $Ass \lor Com_1 \Rightarrow Ass_2$ [69, 121, 130, 134], where Ass is the assumption of $Sys_1 \parallel Sys_2$, Com_1 the commitment of Sys_1, and Ass_2 the assumption of Sys_2, i.e. the most prominent operator is *disjunction*. Essentially, disjunction in the state-based case comes from the use of predicates on state transitions; a transition of $Sys_1 \parallel Sys_2$ is either a transition of Sys_1 or a transition of Sys_2.

State-based Parallel Composition Rule

State-based Process Specifications An assumption/commitment specification of a state-based process P is a tuple (w, As, Co, w'). Then an assumption/commitment specification of P is given by:

$$w \land As \land P \supset Co \land \textit{fin}\ w'$$

which means if w holds initially and any environment action satisfies As, then any action performed by P satisfies Co, and terminates in a state that satisfies *fin* w'. Here w refers to the initial state; As refers to environment actions; Co refers to actions performed by P, i.e. internal actions of the process or program; w' refers to the terminated states (w.r.t the initial state). There are no communications.

The Rule As we described before, a parallel composition rule addresses the premise of $Ass \lor Com_1 \Rightarrow Ass_2$. Following this premise, it can be various forms to be evolved into different rules. We choose the form Cau's work [25], based on Stolen's work [124], and adapted it into an ITL rule. This syntactic parallel composition rule for state-based processes is presented as follows:

$$\frac{\begin{array}{l} w_1 \wedge As_1 \wedge P_1 \supset Co_1 \wedge \textit{fin}\ w'_1 \\ w_2 \wedge As_2 \wedge P_2 \supset Co_2 \wedge \textit{fin}\ w'_2 \\ w \Rightarrow w_1 \wedge w_2 \\ As \vee Co_1 \Rightarrow As_2 \\ As \vee Co_2 \Rightarrow As_1 \\ Co_1 \vee Co_2 \Rightarrow Co \\ \textit{fin}\ w'_1 \wedge \textit{fin}\ w'_2 \Rightarrow \textit{fin}\ w' \end{array}}{w \wedge As \wedge P_1 \parallel P_2 \supset Co \wedge \textit{fin}\ w'}$$

The rule tells that a system, $P_1 \parallel P_2$ can be derived by composing two parallel sub-systems, P_1 and P_2, if and only if all premises of the rule hold.

The first prerequisite is that w_1, As_1, Co_1, and w'_1 must be sound to guarantee the correctness of P_1, and w_2, As_2, Co_2, and w'_2 must be sound to guarantee the correctness of P_2.

The conjunction in, for instance, premise $w \Rightarrow w_1 \wedge w_2$ specifies that the initial state, w, of the composed system, $P_1 \parallel P_2$, must satisfy both the initial state of P_1, w_1, and the initial state of P_2, w_2. The premise, $\textit{fin}\ w'_1 \wedge \textit{fin}\ w'_2 \Rightarrow \textit{fin}\ w'$, indicates that $P_1 \parallel P_2$ terminates ($\textit{fin}\ w'$) if both P_1 and P_2 terminate ($\textit{fin}\ w'_1 \wedge \textit{fin}\ w'_2$).

The disjunction in, for instance, premise $Co_1 \vee Co_2$ can be explained as follows: a state transition from $P_1 \parallel P_2$ is either a state transition from P_1 or a state transition from P_2.

Demonstrative Example

We use a program, $x := y + 1 \parallel_i y := y + 2$, to demonstrate the use of the state-based parallel composition rule. "$\parallel_i$" means the interleaved version of the parallel composition (see Figure 5.1).

```
               1                   2
(x=0,y=0)  --->  (x=1,y=0)  --->  (x=1,y=2)
    |
    |          2                   1
    ---------->  (x=0,y=2)  --->  (x=3,y=2)
```

Figure 5.1: Interleaving

It is easy to see that the following is an invariant:

$$\Box((y = 0 \wedge (x = 0 \vee x = 1)) \vee (y = 2 \wedge (x = 0 \vee x = 1 \vee x = 3)))$$

of the above. This invariant will be used in our As and Co below.

Let P_1 be the program, "$x := y + 1$", and P_2 the program, "$y := y + 2$", then P_1 satisfies

$$w_1 \wedge As_1 \wedge P_1 \supset Co_1 \wedge \textit{fin}\ w_1'$$

$$
\begin{array}{ll}
w_1: & x = 0 \\
As_1: & \Box((y = 0 \wedge (x = 0 \vee x = 1)) \vee \\
 & (y = 2 \wedge (x = 0 \vee x = 1 \vee x = 3)) \vee \\
 & (\bigcirc x = x \wedge \bigcirc y = y)) \\
Co_1: & \Box((y = 0 \wedge (x = 0 \vee x = 1)) \vee \\
 & (y = 2 \wedge (x = 0 \vee x = 1 \vee x = 3)) \vee \\
 & (\bigcirc x = x \wedge \bigcirc y = y)) \\
\textit{fin}\ w_1': & (x = 1 \vee x = 3)
\end{array}
$$

and P_2 satisfies

$$w_2 \wedge As_2 \wedge P_2 \supset Co_2 \wedge \textit{fin}\ w_2'$$

$$
\begin{array}{ll}
w_2: & y = 0 \\
As_2: & \Box((y = 0 \wedge (x = 0 \vee x = 1)) \vee \\
 & (y = 2 \wedge (x = 0 \vee x = 1 \vee x = 3)) \vee \\
 & (\bigcirc x = x \wedge \bigcirc y = y)) \\
Co_2: & \Box((y = 0 \wedge (x = 0 \vee x = 1)) \vee \\
 & (y = 2 \wedge (x = 0 \vee x = 1 \vee x = 3)) \vee \\
 & (\bigcirc x = x \wedge \bigcirc y = y)) \\
\textit{fin}\ w_2': & y = 2.
\end{array}
$$

By the state-based parallel composition rule, $P_1 \| P_2$, i.e., $x := y + 1 \;\|_i\; y := y + 2$ satisfies

$$\begin{array}{ll} w \wedge As \wedge P_1 \| P_2 \supset Co \wedge \mathit{fin}\ w' & \\ w: & x = 0 \wedge y = 0 \\ As: & \Box(\bigcirc x = x \wedge \bigcirc y = y) \\ Co: & \mathit{true} \\ \mathit{fin}\ w': & (x = 1 \vee x = 3) \wedge y = 2. \end{array}$$

5.3 Timing Analysis Guidelines

An important goal of handling time-critical system evolution is to analyse and verify timing properties of the system at run-time. Timing relationships in time-critical systems originate from their requirement to maintain timely interactions with the system's environment. A set of timing parameters have been given in Chapter 2. We consider timing parameters as bounds on computation time, or on the (sub-)intervals (activation period) between start times. Timing analysis at run-time is useful to check the satisfiability of timing behaviours of the system against timing properties specified upon requirements. Furthermore, timing analysis at run-time can be used to evaluate a real system's performance and to identify performance bottlenecks in system design. In our approach, we concentrate on analysing computation times and relationships between different timing (sub-)intervals. This is known as *computation time analysis* in this thesis.

Computation Time Analysis refers to the analysis of timing parameters on computation time of individual or activation periods. In time-critical systems, any computation time must meet the deadline of the computation. The total activation period must meet the total deadline as well. It also involves analysing upper and lower bounds of the average times spent by computations in the process.

Timing parameters are either unary, i.e. timing parameters have bounds on the execution time of a computation, or binary, i.e. timing parameters have bounds on the computation times between the initiation times of two computations [33]. Effectiveness analysis relates to binary timing parameters that define upper/lower bounds on subintervals of computation pairs. Supposing there are two computations, C_1 and C_2, t_1 and t_2 denote start time of C_1 and C_2, respectively. d_1 and d_2 represent computation time of C_1 and C_2, respectively. In general, possible timing relationships between two computations are presented in Table 5.1.

$RelationType$	$Description$
$C_1\ before\ C_2$	$t_1 + d_1 < t_2$
$C_1\ meets\ C_2$	$t_1 + d_1 = t_2$
$C_1\ overlaps\ C_2$	$t_2 - t_1 < d_1$
$C_1\ finishesby\ C_2$	$t_1 + d_1 = t_2 + d_2$
$C_1\ during\ C_2$	$t_1 > t_2\ and\ t_1 + d_1 < t_2 + d_2$
$C_1\ finishes\ C_2$	$t_1 > t_2\ and\ t_1 + d_1 = t_2 + d_2$
$C_2\ overlaps\ C_1$	$t_1 - t_2 < d_2$
$C_2\ meets\ C_1$	$t_1 + d_1 = t_2$
$C_1\ after\ C_2$	$t_2 + d_2 < t_1$
$C_1\ contains\ C_2$	$t_1 < t_2\ and\ d_1 > d_2$
$C_1\ starts\ C_2$	$t_1 = t_2\ and\ d_1 < d_2$
$C_1\ equals\ C_2$	$t_1 = t_2\ and\ d_1 = d_2$
$C_2\ starts\ C_1$	$t_1 = t_2\ and\ d_1 > d_2$

Table 5.1: Possible Binary Relationships between Timing Parameters

During run-time analysis, we will check the various relationships between computations and to check the satisfiability of timing properties. By default, any sequencing dependency between two computations, induces a minimum computation time which must be satisfied in order to meet timing requirements. A minimum computation time (lower bound), l ($l \geq 0$), from computation C_i to computation C_j ($j > i$, $i, j \in N$) can be defined as:

$$t_j \geq t_i + l$$

where t_j is start time of computation C_j and t_i is start time of computation C_i. Similarly, a maximum computation time (upper bound), u ($u \geq 0$) is defined as:

$$t_j \geq t_i + u.$$

In our approach, we use a distinguished state variable, *Time* in ITL, to represent time and let there be an action to increase time, under the following assumptions:

- Time starts at 0. This is defined as: $Time = 0$.
- Time never decreases: $\Box(\bigcirc Time \geq Time)$.

The activation period of an interval can be defined as follows:

$$a = \textit{fin}\ Time$$

where *fin* $Time$ means the final value of the state variable, $Time$, at the end of the interval. Therefore, the computation time, d, of a computation can be expressed as:

$$d = \mathit{fin}\ Time - Time$$

where *fin* $Time$ means the final value of the state variable, $Time$, at the end of the subinterval of the computation and $Time$ is the initial value of $Time$ at the starting point of the computation.

We also consider combined computation times of different computations, i.e. composition of two different sub-intervals. Given two computations, C_1 and C_2, with their computation times, d_1 and d_2, respectively, the combined computation time of C_1 and C_2 can be one of the following three:

- *Sequential Composition*: The combined computation time is represented by $d_1\ ;\ d_2$. Here we use ";" (chop) to express a sequential composition of two computation times. The composed computation time of C_1 and C_2 is defined as:

$$d_1\ ;\ d_2 \ \widehat{=}\ d_1 + d_2;$$

- *Conjoined Parallel Composition*: A conjoined composition of d_1 and d_2 is denoted by $\wedge$ and the conjoined computation time is defined as:

$$d_1 \wedge d_2 \ \widehat{=}\ max(d_1, d_2)$$

where $max(d_1, d_2)$ means that a maximum of the two computation times is chosen to indicate that the time to completion of the concurrent computations is determined by the operation with the largest computation time;

- *Disjoined Parallel Composition*: when the computations, C_1 and C_2, belong to two disjoined processes. This composition is denoted by the symbol, $\vee$, and the combined computation time is defined by a pair, (d_1, d_2), i.e.

$$d_1 \vee d_2 \mathrel{\widehat{=}} (d_1, d_2).$$

  where (d_1, d_2) means that any of the two computation times could be chosen to indicate that the time to completion of the concurrent computations is determined by the operation with the determinable computation, i.e., if one computation determines the whole concurrent process or can terminate both computations, then its computation time is the time to completion of the concurrent computations no matter another computation time is longer or shorter than it.

Furthermore, timing analysis involves the scheduling issues that have been introduced in Chapter 2. The computation times of a process are determined by different schedulers. For further timing analysis, it is not necessary to determine a schedule of processes, but only to verify the *existence* of a schedule. Since there can be many possible schedules, timing analysis proceeds by identifying conditions under which no solutions are possible. A timing parameter is considered inconsistent if it can not be satisfied by *any* implementation at run-time. Inductively, a timing property, consisting of a set of timing parameters, is considered mutually *inconsistent* if these timing parameters can not be satisfied by any implementation at run-time.

5.4 Summary

This chapter gives three key concepts for the handling of time-critical systems' evolution, supporting the integrated approach and guided evolution, given in Chapter 4.

First of all, run-time analysis has been formally defined and explained. This will be used to construct a tool to execute such a run-time analysing process. The second section of this chapter describes compositional rules used to handle various changes in time-critical systems evolution, especially adding or removing sub-systems to/from an existing system, either sequentially or in parallel. We give a brief introduction to composition theory and a short review of its history. A set of compositional rules have been given. They are the general form of the compositional rule, *sequential compositional rule*, and a basic *parallel compositional rule*. A number of simple examples have been given to demonstrate ways of applying these rules. Finally, we give a set of basic notions of timing analysis. A real example of using these notions will be presented in Chapter 7.

Chapter 6

AnaTempura: A Realisation

Objectives:

To describe system architecture of AnaTempura

To give the way of using assertion points in AnaTempura

To discuss run-time analysis realisation by using AnaTempura

To give step-by-step timing analysis by using AnaTempura

To describe visualisation systems in AnaTempura

6.1 System Architecture

AnaTempura is designed as a semi-automatic tool which aims to support the step-by-step methodology described in the thesis. This tool aims at helping engineers in handling the evolution of time-critical systems in a comprehensive way. However, it adopts semi-automation architecture since human intervention is crucial and unavoidable in handling time-critical systems evolution due to the difficulty of understanding a program automatically.

AnaTempura is an integrated workbench for ITL that offers

- specification support,
- validation and verification support in the form of simulation and run-time verification in conjunction with formal specification.

The architecture reflects the framework in Figure 4.4. In this research, the tool has been developed based on the phase-based methodology and is used to support this methodology. AnaTempura is an open architecture that allows new tool components to be easily "plugged in". A significant outcome is to realise the run-time analysis process in AnaTempura, described in Chapter 5. AnaTempura automatically monitors time-critical system execution and analyses the system's run-time behaviours.

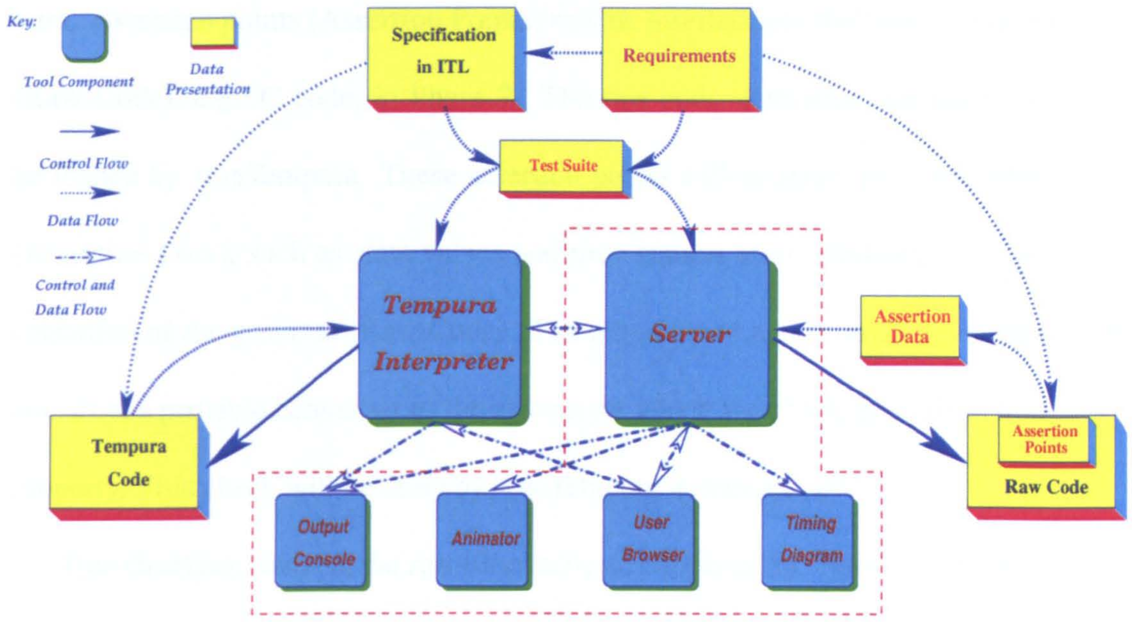


Figure 6.1: General System Architecture of AnaTempura

As shown in Figure 6.1, AnaTempura actually includes all tool components, "Server", "Tempura Interpreter", "Output Console", "Animator", "User Browser", and "Timing

Diagram". Among them, except "Tempura Interpreter", which already exists and will be introduced in a following section, other component (included in a box with dashed lines in Figure 6.1) were either developed from scratch ("Server", "Animator", and "Timing Diagram") or updated for new requirements ("Output Console" and "User Browser"). We will describe each of them one by one in the following sections. Except for the tool component of "Tempura Interpreter", the other tool components are all developed in this research. Referring to Figure 6.1 and Figure 4.4, first of all, we will use results generated from Phase 1, i.e., textural descriptions to change requirements (Requirements). Then in Phase 2.1, we formulate all behavioural properties of interest, such as safety and timeliness. These are presented as ITL formulae (Specification in ITL) and stored in a Tempura file (Tempura Code), which will be loaded by AnaTempura. Assertion points (Assertion Points) will be inserted into the body of the raw code (Raw Code), e.g. C code, in Phase 2. The raw code with assertion points will also be loaded by AnaTempura. These assertion points will generate run-time information (Assertion Data), such as state values and time stamps after AnaTempura triggers the execution of the program (Raw Code). The sequence of assertion data will be used to construct a possible behaviour of the system for which we check the satisfaction of our property. This check will be done by AnaTempura automatically.

This checking (indeed, the run-time analysis happened in Phase 2.2) is done as follows: start the *AnaTempura* and load the Tempura program (Tempura Code). *AnaTempura* will then start the compiled raw code with assertion points. We then start the Tempura program to check that behaviours as constructed from the data of the assertion points, satisfy the properties. We note here that if the properties are not satisfied, AnaTempura will indicate the errors by displaying what is expected and what the current system actually provides. Therefore, the approach is not just a "keep-tracking" approach, i.e. giving the running results of certain properties of the system. By not only capturing the execution results but also comparing them with formal properties, the AnaTempura tool can validate the properties. There is also a facility to animate behaviours and to visualise timing properties.

There are two main components in AnaTempura (shown in Figure 6.1), *Server* and *Tempura Interpreter*. Server sends/receives data to/from various components. Tempura Interpreter is used to execute Tempura files. AnaTempura also offers powerful visualisation function to enhance the ease of operation of the tool and comprehension of the time-critical system evolution processes. We will describe the two tool components and the visualisation function in the following sections.

6.1.1 The Server

The server is an interactive system with a user-friendly interface that allows the user to analyse the time-critical systems. The user can manually enter any predicate using the User Browser (Figure 6.1). We have used Tcl/TK [128] and Expect [76] to build the tool.

The main function of the server is to capture assertion data sent by assertion points. The server then send them to the Tempura interpreter. The Tempura interpreter executes the Tempura file and convert/check the assertion data received and will send the appropriate messages, *pass* or *fail*, in conjunction with where and how the failure happens, to the server. The server displays the received results on the Output Console (Figure 6.1).

6.1.2 Tempura Interpreter

The C-Tempura interpreter [1] was developed originally by Roger Hale [46] and is now maintained by Antonio Cau and Ben Moszkowski. It is an interpreter for executable Interval Temporal Logic formulae. The first Tempura interpreter was programmed in Prolog by Ben Moszkowski, and was operational around December 2, 1983. Subsequently he rewrote the interpreter in Lisp (mid March, 1984). The C-Tempura interpreter was written in early 1985 by Roger Hale at Cambridge University. For more details, we refer the reader to Moszkowski's book [94].

6.1.3 Realisation of Assertion Points Technique

There are two things we need to do to make assertion points work. One is to build assertion points and another is to build mechanisms to collect assertion data sent by assertion points, interpret captured data and deliver them to other different tool components for further processing. Meanwhile, the construction of assertion points should be as simple as possible.

After the generation of the assertion data, the server will capture them one by one immediately. This process is realised by using the multiple process handling mechanism of Expect. As shown in Figure 6.2, the "Data Capture" tool component of the

[1]The name "C-Tempura" comes from the language C that was used to write the interpreter.

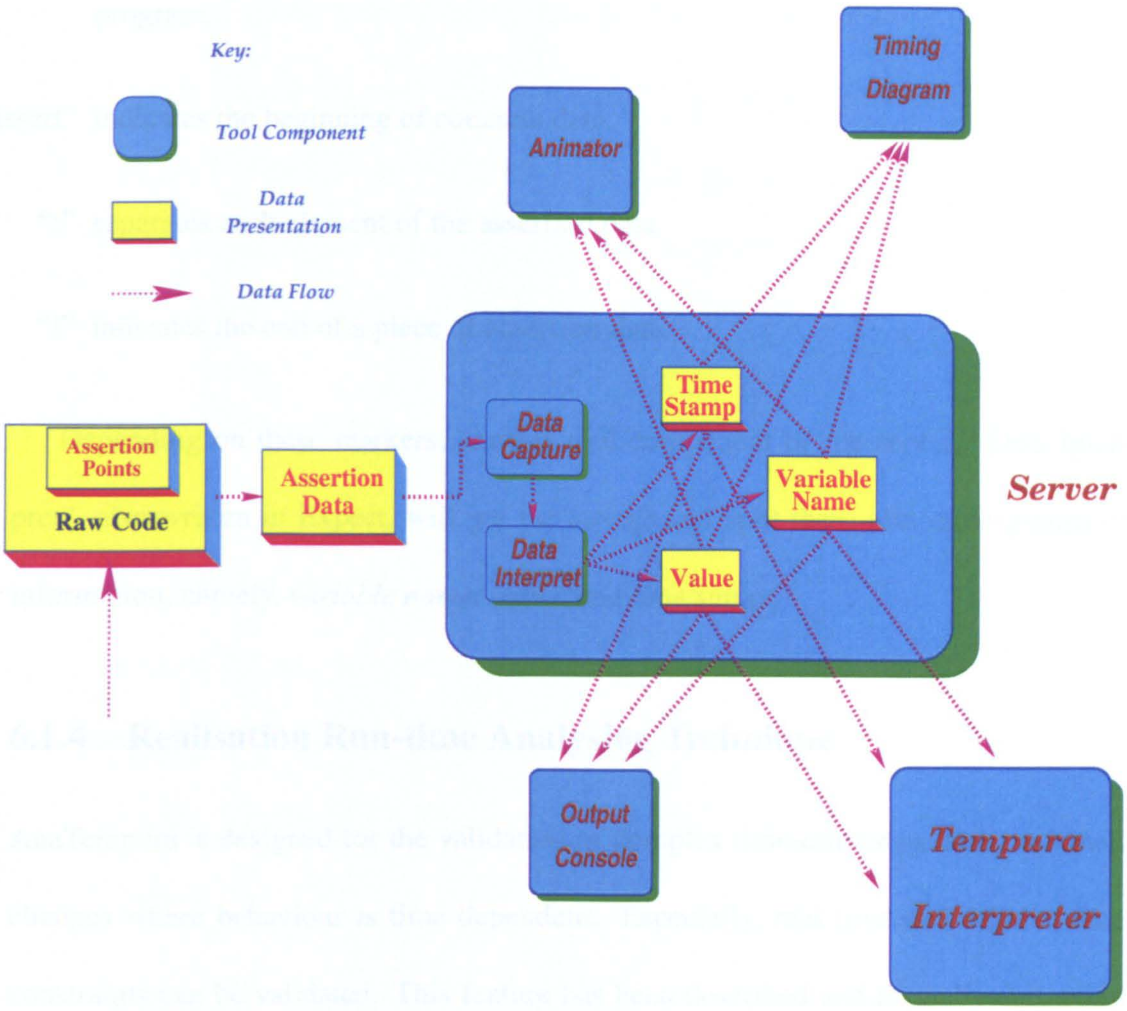


Figure 6.2: Processing Assertion Data

server, written in Expect, will monitor the run of raw code, capture the information sent by assertion points and generate a string, containing assertion data. The form of the string is presented as follows:

!PROG: assert *variable name*: *value*: *time stamp*: !

where a set of markers is included. They are presented as follows.

"!PROG" indicates the beginning of a piece of assertion data and a declaration of the assertion data, for example, here *PROG* means this assertion data is generated from a program.

"assert" indicates the beginning of concrete data.

":" separates each element of the assertion data.

"!" indicates the end of a piece of assertion data.

Depending on these markers, another tool component of the server, "Data Interpret", also written in Expect, will get the strings and split them into three groups of information, namely, *variable name, value*, and *time stamp.*

6.1.4 Realisation Run-time Analysing Technique

AnaTempura is designed for the validation of complex time-critical systems and their changes where behaviour is time dependent. Especially, real time and relative time constraints can be validated. This feature has been described and formally defined in Section 5.1. We will give implementing details in this section.

Recalling the formal specification defined in Section 5.1,

$$Sys \wedge Prop \wedge Monitor$$

AnaTempura triggers and communicates with multiple processes simultaneously. AnaTempura will enable full automation of run-time analysis. Referring to Figure 6.1 and the above formula, we describe them step by step as follows:

- *Invoking/Controlling the run-time analysing process* The server will load and invoke the source code with assertion points. Simultaneously, the Tempura Interpreter will be triggered and used to interpret the Tempura code. In all, as we defined in the above formula, there are three processes that are alive throughout the run-time analysis process, the run of server, the interpreting process of Tempura code and the run of the source code.

- Once the three processes have been triggered, the server controls the other two processes. Monitor will assign an unique identifier to each of two processes, the run of source code and the interpreting process of Tempura so that all data generated by the two processes would not be confused. In addition, the server may trigger some other simultaneous processes, such as animating run-time behaviours and drawing timing diagrams (we will describe this two in following sections).

- *Validating run-time behaviours* For controlling the run of source code, the server will inject pre-set testing data into the process of the source code run. A user can also input instant data through the server during the source code run. That is, the server will prompt a user to input some values through Output Console (Figure 6.1) and a user then inputs required values into User Browser (Figure 6.1). The server can also generate random or special input data in order to test the system under unusual circumstances that rarely happen in the real environment that the system will encounter after its employment. In this case, AnaTempura acts as the environment of the system. The source code will then perform different computations. As we described in Section 5.1 and Section 6.1.3, assertion points will then generate assertion data.

6.2 Realisation of Timing Analysis

Recalling the timing parameters, *start time, computation* (or *execution*) *time, deadline, activation period, communication delays*, and *average times*, we have to specify these parameters in Phase 2.1, i.e., predefining timing properties, and AnaTempura will check whether these parameters holds during the run of the system. The basic idea of run-time analysis is not new. We use a standard method that is to actually run the system under its real working environment, capturing the real execution times and analysing these execution times with pre-defined and verified timing properties. A complete timing diagram will be drawn depending on sequences of values of timing parameters produced at run-time. The timing diagram visualises the run-time timing behaviours of the system. This is a part of the visualisation component of AnaTempura. We will describe it in the next section.

We have given the contents of scheduling in Chapter 2 and timing analysis guidelines in Section 5.3. The following steps take place under timing analysis guidelines and the requirements of scheduling:

- Firstly, calculating from two assertion points, the server calculates the computation time. Then, the server will check the computed computation time with the expected one (specified by a timing property). This check is exactly the same as checking other non-timing variables. Again the results will be "Pass" or "Fail". If the computation time is larger than the required deadline, both the value of the expected and the real computation time will be displayed for further analysis by the server.

- Secondly, a value change file, containing a sequence of changes, will be generated for further use, i.e. the drawing of timing diagrams and further analysis.
- Thirdly, further to deadline checking, the instant computation time will be taken into account in the relationship analysis between different computations. Depending on different analytical targets, we can produce a list of pre-defined relationships between every pair of computations that must be met to fulfil timing properties. Then we can compare the real relationship between each pair of neighbouring or related computations and give comparative results depending on the value change file produced by the server. For example, there is a process, P, containing 10 computations. All computations must satisfy a binary relationship as follows:

$$C_i \; meets \; C_{i+1} : \; t_i + d_i = t_{i+1}$$

where C_i $(0 < i < 9)$ is the ith computation of the process. If any computation does not satisfy the relationship, the system fails

A checking run has been executed and the result is presented as follows:

```
> run test().


!0:: Start

!1:: C_1: Pass time test
```

Computation	Start Time	Computation Time
C_1	0	250
C_2	250	250
C_3	500	70
C_4	570	80
C_5	660	250
C_6	910	250
C_7	1160	70
C_8	1230	80
C_9	1310	250
C_10	1560	250

Table 6.1: An Example of Computation Time List

```
!2:: C_2 : Pass time test

!3:: C_3 : Pass time test

!4:: C_4 : Pass time test

!5:: C_5 : Fail:: Expect: 80, Real: 90

!6:: C_6 : Pass time test

!7:: C_7 : Pass time test

!8:: C_8 : Pass time test

!9:: C_9 : Pass time test

!10:: C_10 : Pass time test

!11:: End


Done!
```

where we can find that the fourth computation does not meet the fifth computation. Therefore, the system fails.

- Fourthly, all run-time values of timing parameters will be taken into account to check whether they satisfy a certain scheduling algorithm. We will produce an extra Tempura file to store these algorithms. The server picks up this Tempura file and checks the correctness of all timing parameters against the scheduling algorithm. At the end, the server will give a form of results similar to the above steps, i.e. the results of "pass" or "fail", and if a parameter fails to pass the check, the server gives two values, one the expected value and the other the real value.

6.3 Visualisation in the Tool

We have given the design of a visualisation system in Section 4.3.2. Recalling Section 4.3.2, we used three ways in the visualisation system. As shown in Figure 6.1, the *textual representation* will mainly be done by the Output Console and the User Browser. *Timing diagrams* will be generated either by the Animator or a Timing Diagram tool component. We will describe the production of timing diagrams later on after we give details of the implementation of the *animation* feature in AnaTempura.

A main contribution of the research is an animation feature in AnaTempura, i.e., visualisation of a program or system's run-time behaviours.

6.3.1 Animation in AnaTempura.

The principle goal of animation in AnaTempura is to help engineers to understand the behaviours of the system which they are working with. The animation shall be made as symbolical representations of hardware, intuitively as meaningful as possible. We design each graphical element to symbolise data attributes expressively. Therefore, we use well known and common metaphors from daily life. We chose the *canvas* widget of TK to construct the animation in AnaTempura. The *canvas* widget provides a general-purpose display that can be programmed to display a variety of objects including arcs, images, lines, ovals, polygons, rectangles, text, and embedded windows. It can be programmed to display a wide variety of objects. Each object can be programmed to respond to user input, or they can be animated under program control. As a result, it provides a very flexible way for developers to visualise run-time behaviours of time-critical systems. More details of *canvas* widget of TK can be found in Welch's book [128].

For portability and little influence on the source code, we built a separate Tcl/Tk file to animate run-time behaviours of the system. The server will load the file and invoke a separate animation window, *Animator* (Figure 6.1). The animator waits to receive processed assertion data sent by the server and displays different animation items on the animation window depending on received data. We will give some examples of animation in Chapter 7.

6.3.2 Timing Diagram in AnaTempura

There are two ways to draw a timing diagram in AnaTempura, one is to use the animation function and the other is to draw timing diagrams via an external software package, GTKWave [44], embedded in AnaTempura. GTKWave is a fully featured GTK+ v1.2 based wave viewer for Unix and Win32. The way of using the animation to draw timing diagrams is suitable to show very simple timing behaviours, e.g., a very small number of timing behaviours in a very short interval. Usually, we use the external GTKWave package to draw timing diagrams.

Timing Diagram Via Animation To draw a timing diagram via animation is the simplest way and displays run-time behaviours of a system instantly. An example has been given in Figure 6.3. Time diagrams are represented by a set of rectangles. The width of rectangles indicates the length of corresponding computation times. The way to draw these rectangles is the same as the animation, described in the previous section. To draw timing diagrams via animation is a real-time visualisation. It takes place during the run-time analysis.

However, because of the limitation of the size of the display window, we can only show the timing diagram sub-interval by sub-interval if the total interval is too long. It is difficult to draw an entire timing diagram of the system in a single window and analyse run-time information under these circumstances.

Timing Diagram Via GTKWave Because to draw timing diagram via animation can not show complex timing behaviours or longer intervals, an alternative way is to

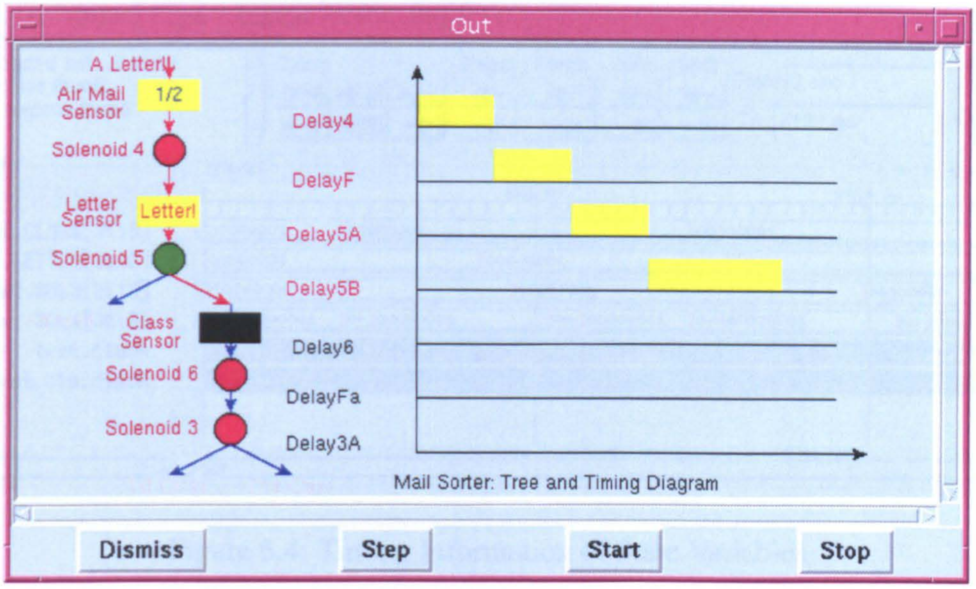


Figure 6.3: Timing Diagram Via Animation

use post-mortem visualisation, i.e. to draw the timing diagram from a trace file that is recorded during the run-time analysis after whole system execution. We have embedded The Veriwell [59] simulator and GTKwave in AnaTempura to produce timing diagrams. Assertion data generated by assertion points is sent to the Veriwell simulator, which produces a *vcd* [2] file, containing timing information, such as names of variables and corresponding time stamps. This vcd file is then used by GTKwave to generate timing diagrams. An example of a timing diagram, drawn by GTKwave, is depicted in Figure 6.4. It shows a diagrammatic representation of timing information for all state variables of interest. As can be seen, various features are available, e.g. setting markers for time lines, change the granularity of time, etc.

[2]Value change diagram (vcd)

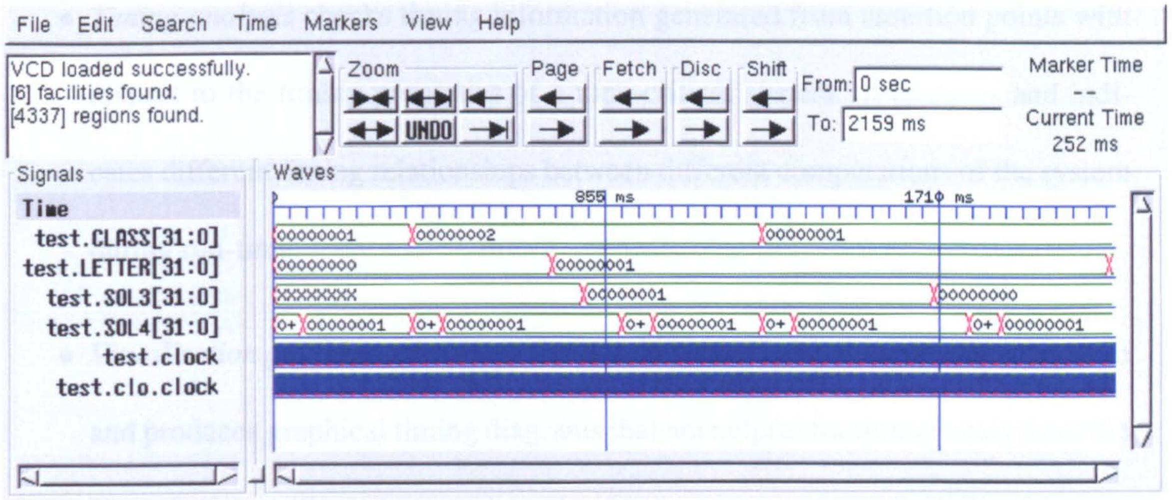

VCD loaded successfully.
[6] facilities found.
[4337] regions found.

Figure 6.4: Timing Information of State Variables

6.4 Summary

This chapter describes a tool, AnaTempura, developed to handle the evolution of time-critical systems following the guidelines given in Chapter 5.

Two main components of AnaTempura, the server and the Tempura Interpreter have been introduced. Four main functions of the server, i.e. the main development of AnaTempura in this thesis, have been described in details. They are: *assertion points*, *run-time analysis*, *timing analysis*, and *visualisation*.

- *Assertion points* generate assertion data at run-time. Assertion data are then used to describe run-time behaviours of a time-critical system.

- *Run-time analysis* triggers a run of a time-critical system, analyses run-time behaviours of the system with respect to the system's specification in Tempura and checks whether the properties of the system which are of interest have been satisfied.

- *Timing analysis* checks timing information generated from assertion points with respect to the timing properties of a time-critical system. It analyses and indicates different timing relationships between different computations of the system during run-time.

- *Visualisation* provides animation of run-time behaviours of a time-critical system and produces graphical timing diagrams that are helpful for timing analysis of the system.

Chapter 7

Case Studies

7.1 Introduction

The two case studies presented in the thesis were selected from several experiments which ranged from small to big scale, un-timed to timed, and sequential to parallel. The first case study is engineering a Post Office letter sorting system, which shows the entire process of managing changes in a time-critical system. It also demonstrated the use of parallel composition rule.

The second case study is a medium size case study following a typical evolutionary development process of an assembly line system. This kind of experiment is much more realistic. The main concern of this case study is to apply the state-based parallel composition rule.

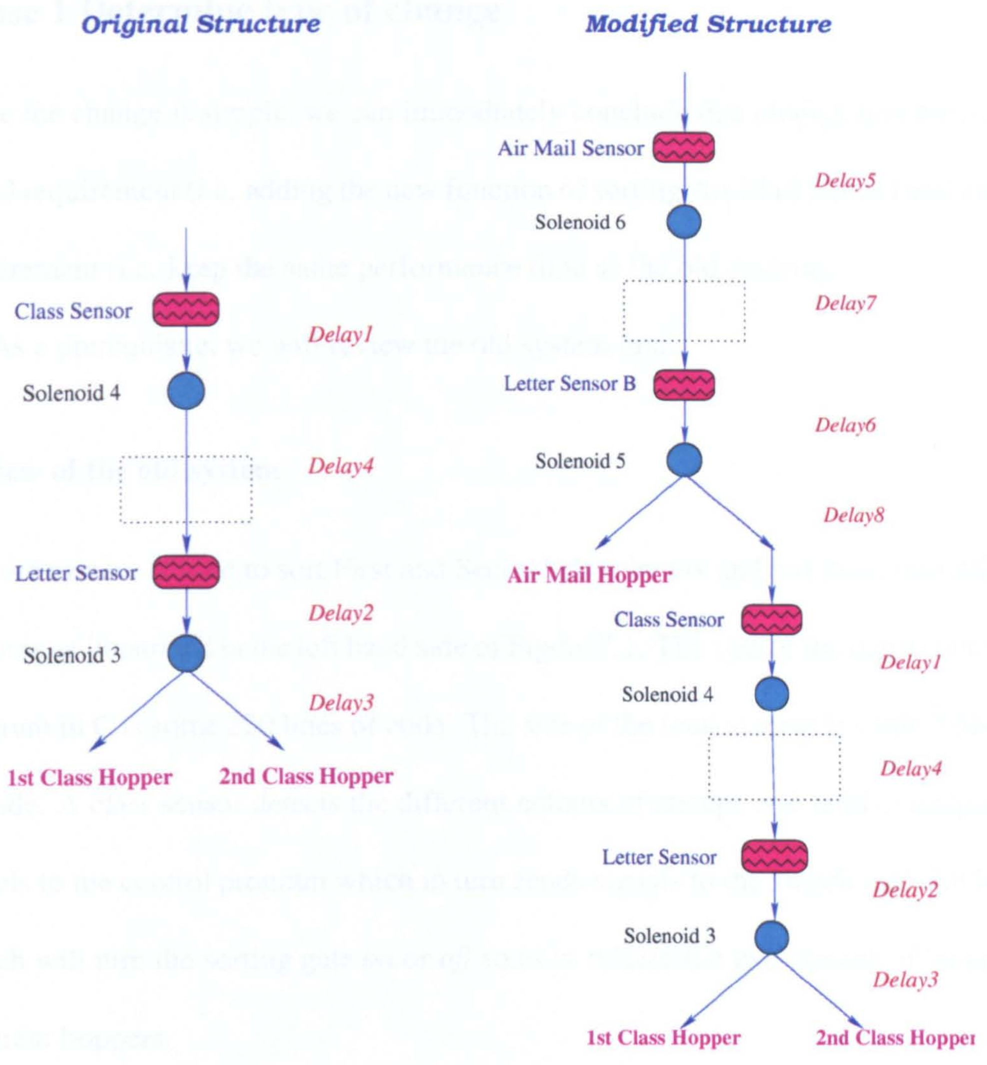


Figure 7.1: The Structure of the Original and Modified Mail Sorter

7.2 Mail Sorter System

Description

This case study is extracted from a Post Office letter sorting system, which is used to sort First and Second Class letters. The decision was made to add a new functionality of processing Air Mail letters. The new sub-system should still sort both First Class and Second Class letters within the original timing constraints.

Phase 1 Determine type of change

Since the change is simple, we can immediately conclude that change involves a functional requirement (i.e. adding the new function of sorting Air Mail letters) and a timing requirement (i.e. keep the same performance time as the old system).

As a prerequisite, we will review the old system first.

Review of the old system

The old system is used to sort First and Second class letters and put them into different hoppers as illustrated in the left hand side of Figure 7.1. The size of the main controlling program in C is some 220 lines of code. The size of the total system is some 2.5K lines of code. A class sensor detects the different colours of stamps and send corresponding signals to the control program which in turn sends signals to the switch (solenoid). The switch will turn the sorting gate *on* or *off* so as to release the two classes of letters into different hoppers.

Delays in the system, which guarantee that sensors and actuators can accomplish their tasks meeting the system's timing requirements, occur at different places as shown in the left hand side of Figure 7.1: Delay1 (70ms) and Delay4 (250ms) for Class sensor and Solenoid 4, respectively; Delay 2 (70ms) and Delay3 (80ms) for Letter sensor and Solenoid 3, respectively.

There are two properties of interests: timeliness and safety properties. The former involves the delay times for switching the solenoids and reading the sensors, whilst the later is a safety requirement: "*no Second class letter in the tray of first class letters and* *visa versa*". These properties are expressed in terms of the Class and Letter Sensors, switches (Solenoid 3 and Solenoid 4), and the various time delays (Delay1, Delay4, Delay2 and Delay3).

The main actuator, Solenoid 3, drops the letter into the correct hopper. If the status of Solenoid 3 is "ON", then it will release the letter into the First class hopper, and when "OFF", it will release the letter into the Second class hopper. All used timing parameters are extracted from the source code and are listed below:

$SensorName$	$ExecutionTime$	$Deadline$
$ClassSensor$	$Delay1$	$70ms$
$LetterSensor$	$Delay2$	$70ms$
$ActuatorName$	$ExecutionTime$	$Deadline$
$Solenoid3$	$Delay3$	$80ms$
$Solenoid4$	$Delay4$	$250ms$

An obvious requirement is that if the Class sensor detects a First Class letter, this letter will eventually be dropped in the First Class hopper within the *activation period* (Delay1 + Delay2 + Delay4 + Delay3) of 470 ms and likewise for a Second Class letter which will be dropped in the second class hopper also within 470ms. This is expressed as follows:

$$
\begin{aligned}
Sorter \;\widehat{=}\; & (\mathit{fin}\ time - time \leqslant 470 \wedge \\
& ((Class\_Sensor = 1st\_Class \supset \Diamond(Solenoid3 = ON)) \wedge \\
& (Class\_Sensor = 2nd\_Class \supset \Diamond(Solenoid3 = OFF)) \\
&) \\
&)^*
\end{aligned}
$$

where "time" is a global variable, indicating the current *time* and "*fin time*", which means the time at the end of the interval. Alternatively, in assumption/commitment form, the old sorter system can also be expressed as:

$$
\begin{aligned}
& (Ass \supset Com) \sqsubseteq Sort \\
& Ass : Letter\_Class = 1st\_Class \vee Letter\_Class = 2nd\_Class \\
& Com : (\mathit{fin}\ time - time \leqslant 470 \wedge \\
& (Class\_Sensor = 1st\_Class \supset \Diamond(Solenoid3 = ON)) \wedge \\
& (Class\_Sensor = 2nd\_Class \supset \Diamond(Solenoid3 = OFF)))
\end{aligned}
$$

The mail sorter system guarantees that it eventually drops a "First Class letter" into the "First Class hopper" and eventually drops a "Second Class letter" into the "Second Class hopper", assuming that its environment sends a first or second class letter.

The timing property is formulated as follows:

$$[\mathit{fin}\ time - time = Delay1 \wedge LetterState = at\_class\_sensor \wedge \textsf{stable}\,(LetterState)]\,;\textsf{skip};$$
$$[\mathit{fin}\ time - time = Delay4 \wedge LetterState = at\_Solenoid\_4 \wedge \textsf{stable}\,(LetterState)]\,;\textsf{skip};$$
$$[\mathit{fin}\ time - time = Delay2 \wedge LetterState = at\_letter\_sensor \wedge \textsf{stable}\,(LetterState)]\,;\textsf{skip};$$
$$[\mathit{fin}\ time - time = Delay3 \wedge LetterState = at\_Solenoid\_3 \wedge \textsf{stable}\,(LetterState)]$$

The timing property expresses the state of a letter and its corresponding deadlines.

Phase 2.1 Impact analysis: Requirement level

After reviewing the old system, we carry out the impact analysis of the required changes to the system. What is required is that the new system should at least satisfy above safety and timeliness properties.

Firstly, we recall the old system. The system should sort every First and Second letter within a period of 470ms. Now we need to give it the ability of sorting Air Mail letters.

The first step is to add the functional requirement to the old system "Sys", changing it to "Sys_1". Then "$Sort$" will be changed to "$\overline{Sort}$", to enable the sorting of Air Mail letters ("$Sort_{Air}$"). The timing requirement "470" will be changed into "470-X". The

"X" is the time needed to sort Air Mail letters.

The system is now:

$\overline{Sort}$:

$(Ass \supset Com) \sqsubseteq \overline{Sort}$

$Ass : Letter\_Class = 1st\_Class \vee Letter\_Class = 2nd\_Class$

$Com : (fin\ time - time \leqslant 470 - X \wedge$

$(Class\_Sensor = 1st\_Class \supset \Diamond(Solenoid3 = ON)) \wedge$

$(Class\_Sensor = 2nd\_Class \supset \Diamond(Solenoid3 = OFF)))$

$$Sys_1 \ \hat{=}\ Sort_{Air}; \overline{Sort}$$

Where $Sort_{Air}$ is

$(Ass_{Air} \supset Com_{Air}) \sqsubseteq Sort_{Air}$

$Ass_{Air} : Letter\_Class = Air \vee Letter\_Class = 1st\_Class \vee Letter\_Class = 2nd\_Class$

$Com_{Air} : (fin\ time - time \leqslant X \wedge$

$(AirMail\_Sensor = Air \supset \Diamond(Solenoid5 = ON)) \wedge$

$(AirMail\_Sensor \neq Air \supset \Diamond(Solenoid5 = OFF)))$

The newly added Air Mail sorter component guarantees that it eventually sends a "Air Mail letter" to the "Air Mail hopper" and eventually sends a "Non-Air Mail letter" to modified the First/Second Class sorter, assuming that the environment sends Air Mail or First or Second Class letters.

The new system is shown in the right hand side of Figure 7.1.

Now we begin to analyse the change in detail with assumption/commitment style of properties and compositional techniques. First of all, we need to recall all the specifications. For ease of description, we also supply a graphical view in Figure 7.2. The graphical description of the old system can be seen in the left hand side of Figure 7.2. The graphical description of the new component can be seen in the right hand side of Figure 7.2.

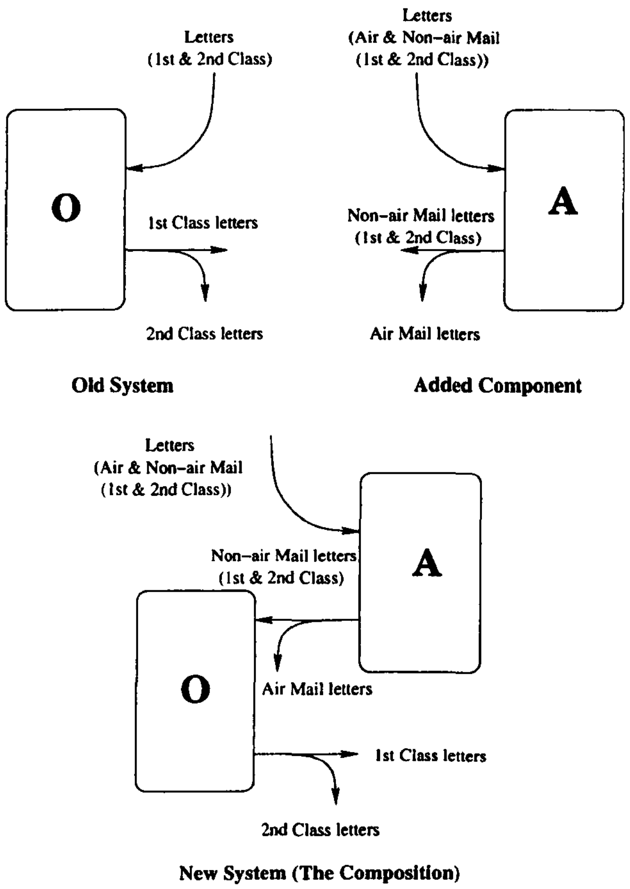


Figure 7.2: The Composition of the Mail Sorter

We add the new component using the following sequential composition rule:

$$\frac{\begin{array}{l}(Ass \supset Com) \sqsubseteq \overline{Sort}, \\ (Ass_{Air} \supset Com_{Air}) \sqsubseteq Sort_{Air}, \\ Ass_C \supset Ass_{Air}, \\ Com_{Air} \supset Ass, \\ Com \supset Com_C\end{array}}{(Ass_C \supset Com_C) \sqsubseteq (Sort_{Air} \,;\, \overline{Sort})}$$

where,

$Ass_C : Letter\_Class = Air \vee Letter\_Class = 1st\_Class \vee Letter\_Class = 2nd\_Class$

$Com_C : \mathit{fin}\ time - time \leqslant 470 \wedge ($

$((AirMail\_Sensor = Air) \supset \Diamond(Solenoid5 = ON)) \wedge$

$((AirMail\_Sensor \neq Air \wedge Class\_Sensor = 1st\_Class) \supset$

$[\Diamond(Solenoid5 = OFF) \wedge \Diamond(Solenoid3 = ON) \wedge$

$((AirMail\_Sensor \neq Air \wedge Class\_Sensor = 2nd\_Class) \supset$

$\Diamond(Solenoid5 = OFF) \wedge \Diamond(Solenoid3 = OFF)]))$

It is easy to see that the premises of the rule hold, i.e., we can apply the rule.

A graphical description of the new system is shown at the bottom Figure 7.2.

In detail, the change needs a new “Air Mail Sensor”, a new “Letter Sensor B”,

SensorName	*ExecutionTime*	*Deadline*
AirMailSensor	*Delay*5	40*ms*
ClassSensor	*Delay*1	40*ms*
LetterSensor	*Delay*2	50*ms*
LetterSensorB	*Delay*6	50*ms*
ActuatorName	*ExecutionTime*	*Deadline*
*Solenoid*6	*Delay*7	90*ms*
*Solenoid*5	*Delay*8	50*ms*
*Solenoid*4	*Delay*4	90*ms*
*Solenoid*3	*Delay*3	60*ms*

Figure 7.3: Timing Information of the Modified System

and two new actuators (Solenoid 6 and Solenoid 5). The reader will notice that the structure of the new system (right hand side of Figure 7.1) has changed to cater for the new addition. As part of the new requirements, the new system should still keep the same process time as the old system. Therefore, after analysing the functional change, we begin to analyse the timing change.

The functional change has introduced new delays (Delay 5, Delay6, Delay7 and Delay8). Among these delays, Delay5 denotes the time for the new Air Mail Sensor to detect the kind of letters and send a relevant signal, Delay7 denotes the execution time for the Solenoid 6 to release letters, Delay6 denotes the time for the new Letter Sensor B to detect the existence of a letter and send a relevant signal, and Delay8 denotes the execution time for the Solenoid 5 to deliver letters into the different hoppers.

The new timing parameters are presented in Figure 7.3. The activation period time for processing a letter is 470ms (Delay1 + Delay4 + Delay2 + Delay3 + Delay5 + Delay7 + Delay6 + Delay8). Therefore, we need to adjust all delays of the old system with respect to new delays. In the new system, the entire interval has been divided into two sub-intervals. Each of two sub-intervals corresponds to one sub-process. The first sub-process is used to sort Air Mail letter, corresponding to "X", and the second one is used to sort First and Second Class letters, corresponding to "470-X". The sequence of timing parameters of second sub-process will follow the same sequence as the old system. They are:

$$Delay5 + Delay7 + Delay6 + Delay8 = X$$

and

$$Delay1 + Delay4 + Delay2 + Delay3 = 470 - X.$$

The new timing property is defined as:

Process for sorting Air Mail letters:

$[\mathit{fin}\ time - time = Delay5 \wedge LetterState =$

$at\_air\_mail\_sensor \wedge \mathsf{stable}\ (LetterState)]$; skip;

$[\mathit{fin}\ time - time = Delay7 \wedge LetterState =$

$at\_Solenoid\_6 \wedge \mathsf{stable}\ (LetterState)]$; skip;

$[\mathit{fin}\ time - time = Delay6 \wedge LetterState =$

$at\_letter\_sensor\_B \wedge \mathsf{stable}\ (LetterState)]$; skip;

$[\mathit{fin}\ time - time = Delay8 \wedge LetterState =$

$at\_Solenoid\_5 \wedge \mathsf{stable}\ (LetterState)]$

Process for sorting 1st/2nd Class letters:

$[\mathit{fin}\ time - time = Delay5 \wedge LetterState =$

$at\_air\_mail\_sensor \wedge \textsf{stable}\ (LetterState)]$; $\textsf{skip}$;

$[\mathit{fin}\ time - time = Delay7 \wedge LetterState =$

$at\_Solenoid\_6 \wedge \textsf{stable}\ (LetterState)]$; $\textsf{skip}$;

$[\mathit{fin}\ time - time = Delay6 \wedge LetterState =$

$at\_letter\_sensor\_B \wedge \textsf{stable}\ (LetterState)]$; $\textsf{skip}$;

$[\mathit{fin}\ time - time = Delay8 \wedge LetterState =$

$at\_Solenoid\_5 \wedge \textsf{stable}\ (LetterState)]$

$[\mathit{fin}\ time - time = Delay1 \wedge LetterState =$

$at\_class\_sensor \wedge \textsf{stable}\ (LetterState)]$; $\textsf{skip}$;

$[\mathit{fin}\ time - time = Delay4 \wedge LetterState =$

$at\_Solenoid\_4 \wedge \textsf{stable}\ (LetterState)]$; $\textsf{skip}$;

$[\mathit{fin}\ time - time = Delay2 \wedge LetterState =$

$at\_letter\_sensor \wedge \textsf{stable}\ (LetterState)]$; $\textsf{skip}$;

$[\mathit{fin}\ time - time = Delay3 \wedge LetterState =$

$at\_Solenoid\_3 \wedge \textsf{stable}\ (LetterState)]$

The analysis results will be collected and documented in the last phase, Phase 3, together with the complete specification of the new system.

Phase 2.2 Impact Analysis: Source Code-level

We derive from the specification of the new system the concrete code using refinement rules. Since developing refinement rules is not a main concern of the thesis, we do not give more details here.

After obtaining the executable code, we use AnaTempura to check it. In the first step, we trigger the checking process for the component of old mail sorter (since it has been modified from the very beginning). Checking runs are generated by AnaTempura (Figure 7.4) following the analysis process depicted in Figure 4.2. The bottom window is a terminal to show outputs and inputs of the running program. The top window shows the checking process. Assertion-points are placed strategically in the code to check the safety property and to the timeliness property. Different designs of assertion-points will be used for different applications. For the mail sorter application, a parameterised C [1] function ("Assertion") is introduced.

```
void assertion(char *aname, int val)  {
    printf("!PROG: assert %s:%d:%d:!\n",
           aname, val, myclock()); }
```

This function sends the state of the system ("aname"), and the current time ("myclock()"[2]). For example, "assertion("class",1)", "assertion ("soloff",SolNo)" and "assertion ("solon",SolNo)" are used to check the safety property of the program, i.e.,

[1]This is due to the application was written in C.
[2]"myclock() is another function which is used to capture the time.

whether a First class can be released into the First class hopper. The following shows a fragment of C code with Assertion Points:

```
/* closing solenoid */ void SolOn(int SolNo) { assertion
("solon",SolNo); }
/* opening solenoid */ void SolOff(int SolNo) { assertion
("soloff",SolNo); }


 if (class_sensor == 1)  {
    /* its yellow - activate solenoid 3 */
   assertion("class",1); SolOff(4); Delay(Dtime1,1);
   SolOn(4); Delay(Dtime4,4);
   scan_sensor ("letter sensor is ?" ,&letter_sensor);
   assertion("lsens",letter_sensor);
   if ( !YellowSet) {
        Delay(Dtime2,2); SolOff(3);  Delay(Dtime3,3);
        YellowSet = 1; } }
```

We note that

```
assertion("class",1); assertion("lsens",letter_sensor);
```

are two assertion-points. They send messages when the code runs. The tool checks and processes the sequence of information, compares it with the properties, and gives messages like "Pass Letter Sensor 1 test" and "Pass Class 1 test", which indicate that some of the safety and the timeliness properties are satisfied.

Then we check the rest of the new system. We have also added corresponding assertion-points for the newly added items. For example, in:

```
if (air_sensor == 1 ) { assertion("air",1); SolOff(6);
     Delay(Dtime5,5); SolOn(6); Delay(Dtime7,7);
     scan_sensor("letter sensor is ?"  ,&letter_sensor);
     assertion("lsens",letter_sensor); if( ! AirSet) {
     Delay(Dtime6,6);  SolOff(5);
     Delay(Dtime8,8);  AirSet = 1; } }
```

where the line

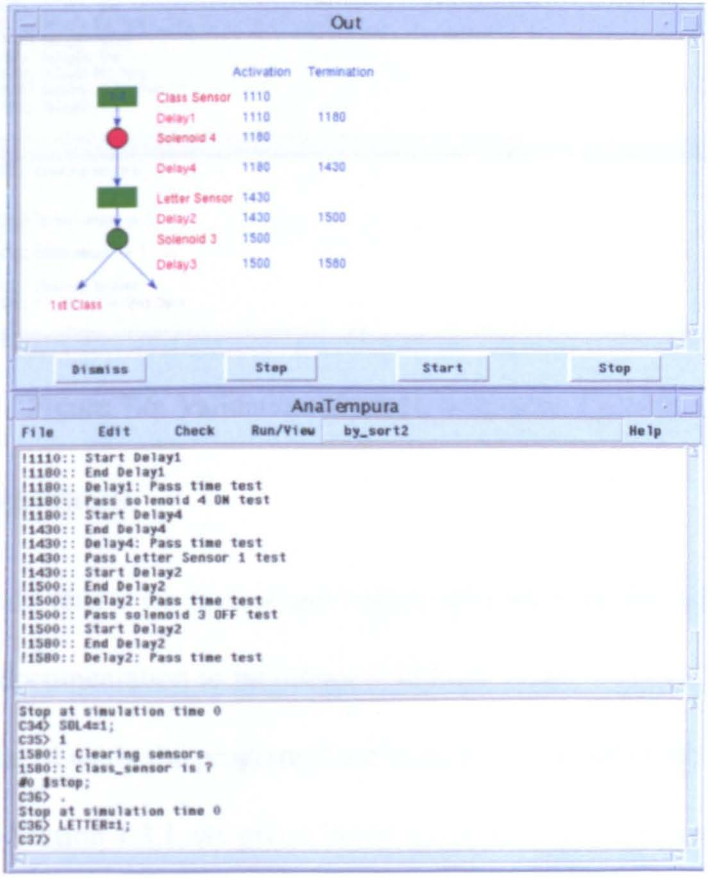
```
assertion("air",1);
```

is used as an assertion-point of checking the process of sorting Air Mail letters. Three assertion-points for the new sensors and actuators relate to the modified safety property, while the three other assertion-points for the new delays relate to the timeliness property. As described before, the time requirement for the change is that the total performance time for processing either Air Mail letter or First and Second Class letter should be the same as the old system (470ms).

Some results from checking the modified system are presented in Figure 7.5, Figure 7.6, and Figure 7.7. They indicate that the safety property has been met for this particular behaviour. The program has sent correct control signals to the actuators, Solenoid 4 and Solenoid 5, and all Air Mail letters have been delivered to the Air Mail Hopper. At the same time, all timeliness properties have been satisfied. The delay times for the sensors and solenoids are correct. We also used some incorrect cases to test the program, such as, we used shorter delays. The checking process reports the error correctly. For example, we restrict the delay (Delay8) of Solenoid 5 to 10. The Solenoid 5 will not have enough time to deliver the letters. We present some test results as follows:

> !70:: Solenoid 5 ON: Pass !70:: Delay8: Start !110:: Delay8: End !110:: Delay8 40: Prog 10 and Real 40.

This tells that the timing property is violated. We then changed the number back to 50ms and tested it again. The check has been passed. We use similar scenarios to test all delays and solenoids.

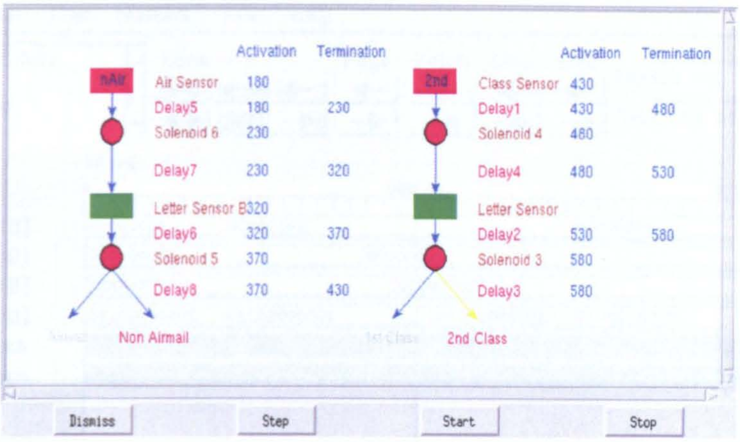


	Activation	Termination
Class Sensor	1110	
Delay1	1110	1180
Solenoid 4	1180	
Delay4	1180	1430
Letter Sensor	1430	
Delay2	1430	1500
Solenoid 3	1500	
Delay3	1500	1580

```
!1110:: Start Delay1
!1180:: End Delay1
!1180:: Delay1: Pass time test
!1180:: Pass solenoid 4 ON test
!1180:: Start Delay4
!1430:: End Delay4
!1430:: Delay4: Pass time test
!1430:: Pass Letter Sensor 1 test
!1430:: Start Delay2
!1500:: End Delay2
!1500:: Delay2: Pass time test
!1500:: Pass solenoid 3 OFF test
!1500:: Start Delay2
!1580:: End Delay2
!1580:: Delay2: Pass time test
```

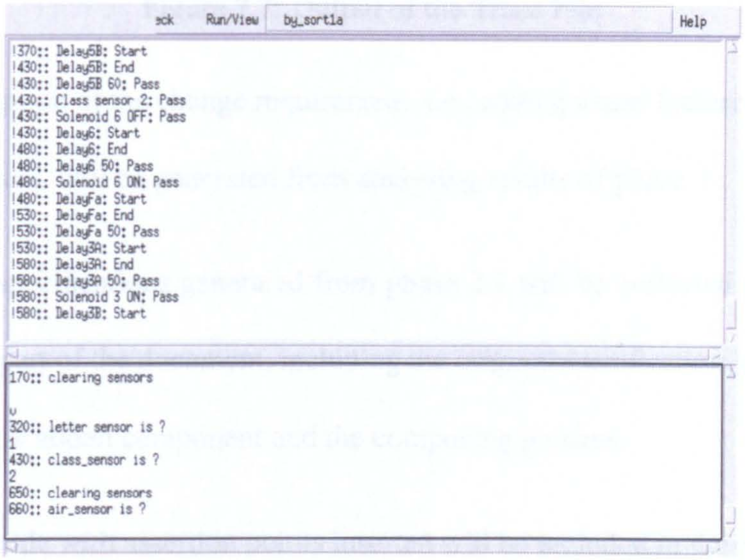
```
Stop at simulation time 0
C34> SOL4=1;
C35> 1
1580:: Clearing sensors
1580:: class_sensor is ?
#0 $stop;
C36> .
Stop at simulation time 0
C36> LETTER=1;
C37>
```

Figure 7.4: Validation of the Original System



```
                    Activation  Termination                   Activation  Termination
Air Sensor          180                      Class Sensor     430
Delay5              180         230          Delay1           430         480
Solenoid 6          230                      Solenoid 4       480
Delay7              230         320          Delay4           480         530
Letter Sensor B320                           Letter Sensor
Delay6              320         370          Delay2           530         580
Solenoid 5          370                      Solenoid 3       580
Delay8              370         430          Delay3           580
```

Figure 7.5: Validation of the New System: Animation



```
!370:: Delay5B: Start
!430:: Delay5B: End
!430:: Delay5B 60: Pass
!430:: Class sensor 2: Pass
!430:: Solenoid 6 OFF: Pass
!430:: Delay6: Start
!480:: Delay6: End
!480:: Delay6 50: Pass
!480:: Solenoid 6 ON: Pass
!480:: DelayFa: Start
!530:: DelayFa: End
!530:: DelayFa 50: Pass
!530:: Delay3A: Start
!580:: Delay3A: End
!580:: Delay3A 50: Pass
!580:: Solenoid 3 ON: Pass
!580:: Delay3B: Start
```

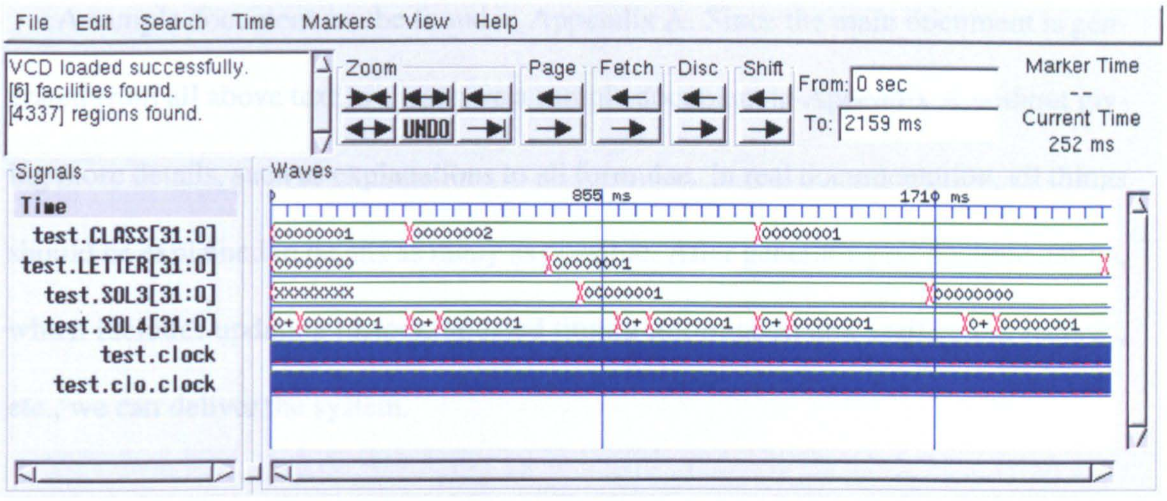
```
170:: clearing sensors

υ
320:: letter sensor is ?
1
430:: class_sensor is ?
2
650:: clearing sensors
660:: air_sensor is ?
```

Figure 7.6: Validation of the New System: Console

Phase 3 Deployment

After having code ready, we review and collect all necessary documents and generate deliverable documentation to be released with the code. Usually, we also need to produce a manual of using the program if the system is a complicated one.

Referring to Section 4.3.1, we give a list of activities which need to be done for this case study as follows.



```
VCD loaded successfully.
[6] facilities found.
[4337] regions found.
```

From: 0 sec
To: 2159 ms
Marker Time --
Current Time 252 ms

Figure 7.7: Output of the Trace File

- A description of the change requirement, i.e., adding a new feature of sorting Air Mail letters, will be generated from analysing results of phase 1.
- Formal specifications generated from phase 2.1 will be collected and written as a major part of the document, including the original specification, a specification of the new added component and the composing process.
- Source code with assertion points inserted will be included in the document.
- Test results of run-time analysis, which are directly saved by AnaTempura, will be included in the document.
- A typical trace file of timing information, which is generated by AnaTempura, will be added into the document, together with some typical screen shots after interpreting this trace file.
- Two typical screen shots will be added into the document, which can be found in this chapter.

A sample document can be found in Appendix A. Since the main document is generated from all above text, we just give a sample document in Appendix A without giving more details, such as explanations to all formulae. In real documentation, all things should be explained in details as many as possible. After generating all documentation, which includes updating history, updated timing information and verification records, etc., we can deliver the system.

7.3 Assembly Line System

Description

In this section we describe a development process of an assembly line system. Initially, the system consists of one robot and one conveyor. A decision was made to add one new robot to the system. The robots and the conveyor will work in parallel. A new control system will be developed from the old controlling systems by parallel composition and some changes.

Review of the old system

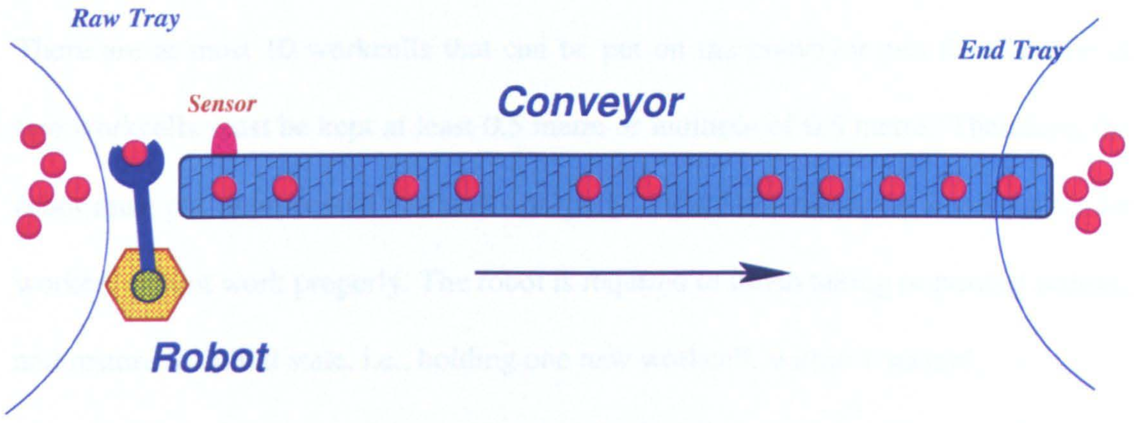


Figure 7.8: Old Assembly Line

First, we describe the system informally. The physical architecture of (an abstract model) the original industrial assembly line system is presented in Figure 7.8. The system consists of one conveyor and one robot. A sensor is installed at the left end of the conveyor. The conveyor carries workcells that are denoted as circles on the conveyor from left to right. Workcells will be processed by workers manually while on the conveyor. The robot takes unprocessed workcells from "Raw Tray" and puts workcells on the conveyor. Processed workcells will drop into an "End Tray". Because the conveyor must cooperate with the robot to finish all tasks, the sensor is responsible for detecting whether there is a workcell at the left end of the conveyor. When the robot is putting a workcell, the conveyor must be stopped. If the sensor sends a signal, indicating there is no workcell at the beginning position of the conveyor and the robot holds a workcell, then control motor of the conveyor will stop and let the robot put a workcell onto the beginning position of the conveyor. If the sensor indicates there is a workcell at the left end of the conveyor, then the conveyor runs and the robot waits.

In addition, the working length of the conveyor is 5 metres. Depending on the working speed of workers, the running speed of conveyor is 0.5 metre per second. There are at most 10 workcells that can be put on the conveyor and the distance of two workcells must be kept at least 0.5 metre or multiple of 0.5 metre. Therefore, the robot must put unprocessed workcells properly. Otherwise either the conveyor or the workers cannot work properly. The robot is required to finish taking or putting actions, and restore its initial state, i.e., holding one new workcell, within 1 second.

In the rest of the reviewing process, we will give the formal specification of the system requirements and design. First, we give state variables and notations for specifying the system. The conveyor can be expressed by an array, L, with ten Boolean variables, indicating ten positions of the conveyor. $L[i](0 \leq i \leq 9)$ denotes the i-th position of the conveyor from the left side. If $L[i] = 1$, then there is a workcell at the i-th position. Otherwise, $L[i] = 0$ means the position is empty. The length of the list L is defined as $\mid L \mid$. A Boolean variable S is used to indicate the current state of the conveyor. If $S = 1$, then the conveyor is in the stopped state. Otherwise, the running state of the conveyor will be denoted by $S = 0$.

The state of sensor can be expressed by $L[0]$. That is, if $L[0] = 1$, then there is a workcell on the left end of the conveyor and the sensor should send a signal, reporting this situation.

Furthermore, a Boolean variable, R, are set to express the status of the robot that whether a robot holds a workcell in its arms/hands or not. If $R = 1$, the robot holds a workcell in its hand at current state. Otherwise, $R = 0$ indicates that the robot holds nothing.

Based on the analysis, we can explore several different cases. The conveyor must be in the stopped state in some cases and can move in the other cases. All these cases guarantee the correct running of the system. First of all, we define the states of **move** and **stop**.

The state of **move** can be defined as:

$$MOVE: \bigcirc \bigwedge_{i=0}^{8} L[i] = L[i-1]$$

The state of **stop** can be defined as:

$$STOP: \bigcirc \bigwedge_{i=1}^{8} L[i] = L[i]$$

We consider the robot and the conveyor as two parallel processes of the system. The system is expressed as $Robot||Conveyor$. Therefore, a set of possible states of the system can be defined as follows:

① $R = 0 \wedge L[0] = 0$

② $R = 1 \wedge L[0] = 0$

③ $R = 0 \wedge L[0] = 1$

④ $R = 1 \wedge L[0] = 1$

where, $R = 0$ means that the robot does not hold any work cell whilst $R = 1$ indicates that the robot holds a workcell. If $L[0] = 1$, then there is a workcell on the left end of the conveyor and the sensor should report it by sending a signal. $L[0] = 0$ indicates the left end of the conveyor is empty. We use the symbols,①, ②,③, and ④, to represent four different cases, respectively, for following specifications.

Following our methodology, we define control systems of the robot and the conveyor in assumption/commitment form one by one.

Firstly, we analyse the above four different cases with respect to the control system of the conveyor. It is easy to see that the conveyor can not run when the cases ②, because the conveyor should give the robot a chance to put a workcell on its left end. For the rest of cases conveyor can move since the robot can put nothing on the left end of the conveyor. In result, the control system of the conveyor can be defined as:

$$w_{conv} \wedge As_{conv} \wedge Conveyor \supset Co_{conv} \wedge \mathit{fin}\ w'_{conv}$$

$$\begin{array}{ll}
w_{conv}: & S = 1 \wedge \bigwedge_{i=0}^{9} L[i] = 0 \\
As_{conv}: & \mathit{true} \\
Co_{conv}: & \textcircled{1} \vee \textcircled{3} \vee \textcircled{4} \supset [\bigcirc S = 0 \wedge MOVE] \wedge \\
 & \textcircled{2} \supset [\bigcirc S = 1 \wedge STOP] \\
 & \wedge \bigcirc Time = Time + 1 \\
\mathit{fin}\ w'_{conv}: & \mathit{true}
\end{array}$$

where $\bigcirc Time = Time + 1$ gives the timing property of the conveyor control system. That is, all operations must be performed in one time unit (1 second).

We then analyse each case for the robot. For cases, ①, or ③, the robot holds nothing. It should get a workcell immediately, but without affecting the running state of the conveyor. For the case ②, the robot holds a workcell and there is no workcell on either end of the conveyor. In this case, the conveyor stops so that the robot can put one workcell on the left end of the conveyor. For the case ④, because there is a workcell on the left end of the conveyor and the robot cannot put any new workcell, then the robot will keep holding the workcell without taking any action. The control system of the robot can then be defined as:

$$w_R \wedge As_R \wedge \mathit{Robot} \supset Co_R \wedge \mathit{fin}\ w'_R$$

$$\begin{array}{ll}
w_R: & R = 0 \\
As_R: & \mathit{true} \\
Co_R: & ② \supset [\bigcirc L[0] = 1 \wedge \bigcirc R = 0] \wedge \\
 & [① \vee ③ \vee ④] \supset \bigcirc R = 1 \\
 & \wedge\ \bigcirc Time = Time + 1 \\
\mathit{fin}\ w'_R: & true
\end{array}$$

where the timing property, $\bigcirc Time = Time + 1$ has the same meaning as the one in the conveyor control system.

We compose the robot control system and the conveyor control system together using the following state-based parallel composition rule to get the system, Sys, i.e., $(\mathit{Robot} \| Conveyor)$.

$$\begin{array}{|l|}
\hline
w_R \wedge As_R \wedge Robot \supset Co_R \wedge fin\ w'_R \\
w_{conv} \wedge As_{conv} \wedge Conveyor \supset Co_{conv} \wedge fin\ w'_{conv} \\
w_{Sys} \Rightarrow w_R \wedge w_{conv} \\
As_{Sys} \vee Co_R \Rightarrow As_{conv} \\
As_{Sys} \vee Co_{conv} \Rightarrow As_R \\
Co_R \vee Co_{conv} \Rightarrow Co_{Sys} \\
fin\ w'_R \wedge fin\ w'_{conv} \Rightarrow fin\ w'_{Sys} \\
\hline
w_{Sys} \wedge As_{Sys} \wedge Robot \| Conveyor \supset Co_{Sys} \wedge fin\ w'_{Sys} \\
\hline
\end{array}$$

We can then generate a specification after composing the robot control system and the conveyor control system together. It is:

$$w_{Sys} \wedge As_{Sys} \wedge Robot \parallel Conveyor \supset Co_{Sys} \wedge fin\ w'_{Sys}$$

$$\begin{array}{ll}
w_{Sys}: & R = 0 \wedge S = 1 \wedge \bigwedge_{i=0}^{9} L[i] = 0 \\
As_{Sys}: & true \\
Co_{Sys}: & (\textcircled{1} \vee \textcircled{3} \vee \textcircled{4} \supset [\bigcirc S = 0 \wedge MOVE] \wedge \\
 & \textcircled{2} \supset [\bigcirc S = 1 \wedge STOP]) \vee \\
 & (\textcircled{2} \supset [\bigcirc L[0] = 1 \wedge \bigcirc R = 0] \wedge \\
 & [\textcircled{1} \vee \textcircled{3} \vee \textcircled{4}] \supset \bigcirc R = 1) \\
 & \wedge \bigcirc Time = Time + 1 \\
fin\ w'_{Sys}: & true.
\end{array}$$

This specification tells that a transition of Sys is either a transition of $Robot$ or a transition of $Conveyor$ during a state in this state-based case. In a certain state, a commitment of the Sys will be either a commitment of the robot or a commitment of the conveyor.

Phase 1 Determine type of change

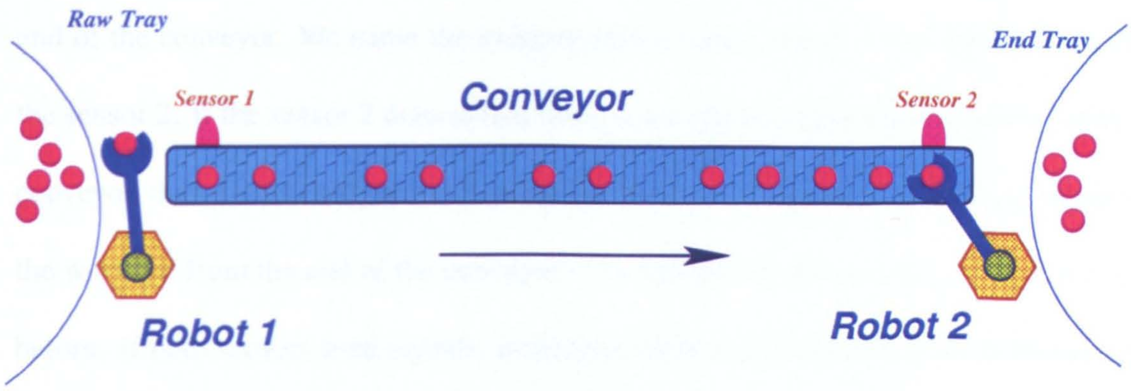


Figure 7.9: New Assembly Line

The physical architecture of (an abstract model) the new assembly line system is presented in Figure 7.9. A new robot and a corresponding, new sensor have been added into the new system. A new robot control program will be added into the assembly control system. Both robots and the conveyor run in parallel. Therefore, the change involves functional requirements of adding a new robot control component and a new variable of storing the signal sent by the new sensor, as well as a timing requirement of limiting the new robot's action time.

Phase 2.1 Impact analysis: Requirement level

Firstly, we analyse the difference between the old system and the new system. Comparing to the old system, instead of dropping into the "End Tray", a processed workcell will be taken by the new robot and put into the "End Tray". We name the existing robot as the robot 1 and the new robot as the robot 2. When both robots are taking or putting workcells, the conveyor must be stopped. The new sensor will be installed at the right end of the conveyor. We name the existing sensor as the sensor 1 and the new one as the sensor 2. If the sensor 2 detects that there is a workcell at the right most end of the conveyor, then it sends a signal, indicating that the robot 2 should take action to remove the workcell from the end of the conveyor. The function of the sensor 1 is the same as before. If both sensors send signals, indicating there is no workcell at the both end of the conveyor, then the conveyor can run and two robots wait.

The state of the sensor 2 can be expressed by $L[9]$. That is, if there is a workcell on the right end of the conveyor and the sensor 2 should send a signal, reporting this situation by assigning a value 1 to $L[9]$.

Furthermore, two boolean variables, $R1$ and $R2$, are set to express the status of the two robots that whether a robot holds a workcell in its arms/hands or not. If $R1 = 1$, the robot 1 holds a workcell in its hand at current state. Otherwise, $R1 = 0$ indicates that the robot 1 holds nothing.

A safety property should be satisfied. That is, the robot 2 must always take away a workcell at the right end of the conveyor before the workcell reaches the end. Otherwise, the processed workcell will fall and break. The safety property, namely, $WorkcellSafe$, must be satisfied. It is specified as follows in ITL:

$$WorkcellSafe \mathrel{\widehat{=}} (\Box \Diamond \neg (L[9] = 1 \wedge R2 = 1))$$

which means that the power system will be damaged and a catastrophic result could be happened if the system enters a state that a workcell reaches the right end of the conveyor and the robot 2 still holds a workcell and is not able to take a new workcell.

Based on the analysis, we can explore several different cases as we did in the reviewing stage. The conveyor must be in the stopped state in some cases and can move in the other cases. All these cases guarantee the correct running of the system. First of all, we define the states of **move** and **stop**.

The state of **move** can be defined as:

$$MOVE: \ \bigcirc \bigwedge_{i=0}^{8} L[i] = L[i-1]$$

The state of **stop** can be defined as:

$$STOP: \ \bigcirc \bigwedge_{i=1}^{8} L[i] = L[i]$$

As we can see, the above definitions are the same as ones defined in the old system. Differing from the old system, we need to consider a new variable, $L[9]$, corresponding to the new sensor 2. Therefore, we have eight cases of possible states of the system. They are:

① $R_1 = 0 \wedge L[0] = 0 \wedge L[9] = 0$

② $R_1 = 1 \wedge L[0] = 0 \wedge L[9] = 0$

③ $R_1 = 0 \wedge L[0] = 1 \wedge L[9] = 0$

④ $R_1 = 1 \wedge L[0] = 1 \wedge L[9] = 0$

⑤ $R_1 = 0 \wedge L[0] = 1 \wedge L[9] = 1$

⑥ $R_1 = 1 \wedge L[0] = 1 \wedge L[9] = 1$

⑦ $R_1 = 1 \wedge L[0] = 0 \wedge L[9] = 1$

⑧ $R_1 = 0 \wedge L[0] = 0 \wedge L[9] = 1$

where, $R_1 = 0$ means that the robot 1 does not hold any work cell whilst $R_1 = 1$ indicates that the robot 1 holds a workcell. $L[0]$ and $L[9]$ denote the state of the sensor 1 and the sensor 2, respectively. That is, if $L[0] = 1$, then there is a workcell on the left end of the conveyor and the Sensor 1 should report it by sending a signal. $L[0] = 0$ or $L[9] = 0$ indicates either the left end of the conveyor is empty or the right end of the conveyor is empty. We use the symbols,①, ②,③, ④, ⑤, ⑥, ⑦, and ⑧, to represent eight different cases, respectively, for following specifications.

Following our methodology, we define control systems of two robots and the conveyor in assumption/commitment form one by one. Because of the new four cases, we need to reconsider all specifications of two robots and the conveyor. Firstly, we analyse the above eight different cases with respect to the control system of the conveyor. It is easy to see that the conveyor can not run when cases, ⑤, ⑥, ⑦, and ⑧. Because there is a workcell on the right end of the conveyor, if the conveyor moves, then the workcell falls. For cases ① and ③, the conveyor can move since there is no workcell on the right end of the conveyor and the robot 1 can put nothing on the left end of the conveyor. For case ④, although the robot holds a workcell and is ready to put it on the left end of the conveyor, but the left end of the conveyor has not been vacated yet, therefore, the conveyor should keep moving to vacate its left end. Only for case ②, it is necessary for the conveyor to stop and wait for a new workcell put by the robot 1 on its left end. In result, the control system of the conveyor can be defined as:

$$w_{conv} \wedge As_{conv} \wedge Conveyor \supset Co_{conv} \wedge \mathit{fin}\ w'_{conv}$$

$$\begin{array}{ll}
w_{conv}: & S = 1 \wedge \bigwedge_{i=0}^{9} L[i] = 0 \\
As_{conv}: & \mathit{true} \\
Co_{conv}: & ① \vee ③ \vee ④ \supset [\circ S = 0 \wedge MOVE] \wedge \\
 & ② \vee ⑤ \vee ⑥ \vee ⑦ \vee ⑧ \supset [\circ S = 1 \wedge STOP] \\
 & \wedge \circ Time = Time + 1 \\
\mathit{fin}\ w'_{conv}: & \mathit{true}
\end{array}$$

where $\circ Time = Time + 1$ gives the timing property of the conveyor control system. That is, all operations must be performed in one time unit (1 second). In the cases of ① and ③, the conveyor keeps running. In the cases of ②, ④,or ⑤, the conveyor will stop to wait for two robots' operations. As we can see, the general specification of the conveyor is the same as the one in the old system, but with four new cases. The change of the original specification is rather trivial.

We then analyse each case for the robot 1. For cases, ①, ③, ⑤,or ⑧, the robot 1 holds nothing. It should get a workcell immediately, but without affecting the running state of the conveyor. For case ②, the robot 1 holds a workcell and there is no workcell on either end of the conveyor. In this case, the conveyor is required to stop so that the robot 1 can put one workcell on the left end of the conveyor. Similar to case ②, for case ⑦, it is required that not only the robot 1 should put one workcell on the left end of the conveyor, but also a workcell on the right end of the conveyor should be removed. In result, the conveyor must stop. For cases, ④ or ⑥, because there is a workcell on the left end of the conveyor and the robot 1 cannot put any new workcell, then the robot 1 will keep holding the workcell without taking any action. The control system of the robot 1 can then be defined as:

$$w_{R1} \wedge As_{R1} \wedge Robot1 \supset Co_{R1} \wedge \mathit{fin}\ w'_{R1}$$

w_{R1} : $R_1 = 0$

As_{R1} : *true*

Co_{R1} : $[② \vee ⑦] \supset [\bigcirc L[0] = 1 \wedge \bigcirc R_1 = 0] \wedge$

$[① \vee ③ \vee ④ \vee ⑤ \vee ⑥ \vee ⑧] \supset \bigcirc R_1 = 1$

$\wedge\ \bigcirc Time = Time + 1$

$\mathit{fin}\ w'_{R1}$: *true*

where the timing property, $\bigcirc Time = Time + 1$ has the same meaning as the one in the conveyor control system. In the cases of ② or ⑤, when the conveyor stops, if Robot 1 holds a workcell, it will then put the workcell to the left end of the conveyor whilst Robot 1 will keep idle. In the cases of ③ and ④, because the conveyor runs, Robot 1 takes no action. Similarly, we can find that we applied some small changes to the original specification of the robot (1), without changing the general form of the specification.

We apply the following state-based parallel composition rule to get the system, Sys, i.e., $(Robot1 \| Conveyor)$.

$$\boxed{\begin{array}{l}
w_{R1} \wedge As_{R1} \wedge Robot1 \supset Co_{R1} \wedge \mathit{fin}\ w'_{R1} \\
w_{conv} \wedge As_{conv} \wedge Conveyor \supset Co_{conv} \wedge \mathit{fin}\ w'_{conv} \\
w_{Sys} \Rightarrow w_{R1} \wedge w_{conv} \\
As_{Sys} \vee Co_{R1} \Rightarrow As_{conv} \\
As_{Sys} \vee Co_{conv} \Rightarrow As_{R1} \\
Co_{R1} \vee Co_{conv} \Rightarrow Co_{Sys} \\
\mathit{fin}\ w'_{R1} \wedge \mathit{fin}\ w'_{conv} \Rightarrow \mathit{fin}\ w'_{Sys} \\
\hline
w_{Sys} \wedge As_{Sys} \wedge Robot1 \| Conveyor \supset Co_{Sys} \wedge \mathit{fin}\ w'_{Sys}
\end{array}}$$

We can then generate a specification after composing the robot 1 control system and the conveyor control system together. It is:

$$w_{Sys} \wedge As_{Sys} \wedge Robot1 \parallel Conveyor \supset Co_{Sys} \wedge fin\ w'_{Sys}$$

$$\begin{array}{ll}
w_{Sys}: & R_1 = 0 \wedge S = 1 \wedge \bigwedge_{i=0}^{9} L[i] = 0 \\
As_{Sys}: & true \\
Co_{Sys}: & (① \vee ③ \vee ④ \supset [\bigcirc S = 0 \wedge MOVE] \wedge \\
& ② \vee ⑤ \vee ⑥ \vee ⑦ \vee ⑧ \supset [\bigcirc S = 1 \wedge STOP]) \vee \\
& ([② \vee ⑦] \supset [\bigcirc L[0] = 1 \wedge \bigcirc R_1 = 0] \wedge \\
& [① \vee ③ \vee ④ \vee ⑤ \vee ⑥ \vee ⑧] \supset \bigcirc R_1 = 1) \\
& \wedge \bigcirc Time = Time + 1 \\
fin\ w'_{Sys}: & true.
\end{array}$$

Through analysing the above specifications and the composing process, we can easily find that we simply repeated our work as in the reviewing stage. We do not actually change anything significantly. However, we can find that the change, i.e., introducing a new variable, affects all existing components of the system. Apparently, the use of the compositional rule reduces the work of analysing and specifying the changes.

Then we consider the robot 2 and the whole assembly system. We can treat the whole assembly system as a composition between the system, Sys ($Robot1 || Conveyor$), and the robot 2 control system in all. As we can see that Sys does not use R2 because it is a state variable of the robot 2. The shared variable between Sys and the robot2 is L[9] (and Time). Based on these facts, we can specify the robot 2 as follows:

$$w_{R2} \wedge As_{R2} \wedge Robot2 \supset Co_{R2} \wedge fin\ w'_{R2}$$

$$\begin{array}{ll}
w_{R2}: & R_2 = 0 \\
As_{R2}: & true \\
Co_{R2}: & R_2 = 1 \supset \bigcirc R_2 = 0 \\
 & L[9] = 1 \wedge R_2 = 0 \supset \bigcirc R_2 = 1 \\
 & L[9] = 0 \wedge R_2 = 0 \supset \bigcirc R_2 = 0 \\
 & \wedge\ \bigcirc Time = Time + 1 \\
fin\ w'_{R2}: & true
\end{array}$$

where the timing property, $\bigcirc Time = Time + 1$, is still the same as ones in the conveyor control system and the robot 2 control system. $R_2 = 1 \supset \bigcirc R_2 = 0$ means that if the robot 2 holds a workcell, it must put the workcell to the tray in 1 second. If there is a workcell on the right end of the conveyor ($L[9] = 1$) and the robot 2 holds nothing ($R_2 = 0$), Robot 2 must take the workcell from the right end of the conveyor and put it into a tray in 1 second. Otherwise, Robot 2 takes no action.

Secondly, we compose the robot 2 control system and the Sys together using the following state-based parallel composition rule. We use "$Assembly$" ($Robot1 || Conveyor$ ||

*Robot*2) to represent the final assembly control system.

$$\frac{\begin{array}{l} w_{Sys} \wedge As_{Sys} \wedge Robot1 \| Conveyor \supset Co_{Sys} \wedge fin\ w'_{Sys} \\ w_{R2} \wedge As_{R2} \wedge Robot2 \supset Co_{R2} \wedge fin\ w'_{R2} \\ w \Rightarrow w_{Sys} \wedge w_{R2} \\ As \vee Co_{Sys} \Rightarrow As_{R2} \\ As \vee Co_{R2} \Rightarrow As_{Sys} \\ Co_{Sys} \vee Co_{R2} \Rightarrow Co \\ fin\ w'_{R1C} \wedge fin\ w'_{R2} \quad \Rightarrow fin\ w' \end{array}}{w \wedge As \wedge Robot1 \| Conveyor \| Robot2 \supset Co \wedge fin\ w'}$$

We can then generate a specification after composing the robot 2 control system and the Sys together. It is:

$$w \wedge As \wedge Assembly \supset Co \wedge fin\ w'$$

$$
\begin{array}{ll}
w: & R_1 = 0 \wedge S = 1 \wedge \bigwedge_{i=0}^{9} L[i] = 0 \wedge R_2 = 0 \\
As: & true \\
Co: & ((① \vee ③ \vee ④ \supset [\bigcirc S = 0 \wedge MOVE] \wedge \\
 & ② \vee ⑤ \vee ⑥ \vee ⑦ \vee ⑧ \supset [\bigcirc S = 1 \wedge STOP]) \vee \\
 & ([② \vee ⑦] \supset [\bigcirc L[0] = 1 \wedge \bigcirc R_1 = 0] \wedge \\
 & [① \vee ③ \vee ④ \vee ⑤ \vee ⑥ \vee ⑧] \supset \bigcirc R_1 = 1) \vee \\
 & (R_2 = 1 \supset \bigcirc R_2 = 0 \\
 & L[9] = 1 \wedge R_2 = 0 \supset \bigcirc R_2 = 1 \\
 & L[9] = 0 \wedge R_2 = 0 \supset \bigcirc R_2 = 0) \\
 & \wedge \bigcirc Time = Time + 1 \\
fin\ w': & true
\end{array}
$$

where,

$$Assembly \mathrel{\widehat{=}} Robot1 \parallel Conveyor \parallel Robot2$$

It is also easy to see that the premises of the rule hold.

So far, we can find the advantage of using compositional theory. Towards this example, there is no need to modify the original specification significantly or build the specification from scratch totally. We only need to do slight change to the original system and add a new component via applying the compositional rule. This greatly saves time. In this case study, we generally reused two original specifications and developed only one new specification.

After impact analysis at the requirement level, we derive from the specification of the new system the concrete code using refinement rules and generate the executable code. The rest phases are similar to those presented in Section 7.2. A set of assertion points have been designed and inserted into the executable code. We run the AnaTempura and analyse all run-time behaviours of the system. The run-time analysis give us some very satisfactory results. The evolution of the assembly line system has been proved successful and the new system has been delivered, meeting all requirements.

Phase 2.2 Impact Analysis: Source Code-level

In the case study, we do not refine the code into any specific language. We derive from the specification to a runnable Tempura code instead. A user can easily convert the runnable Tempura code into a system in any other programming language, such as C or Java. This shows the language-independent feature in our approach.

At the beginning of this phase, we decided the property of interest, which we are going to check, is the safety property, i.e.,

$$WorkcellSafe \;\widehat{=}\; (\Box \Diamond \neg (L[9] = 1 \wedge R2 = 1)).$$

We need always check the values of $L[9]$ and $R2$ in each state and validate that the values of $L[9]$ and $R2$ will not equal to 1 at the same time.

We then firstly derive a runnable Tempura code. As an example, we give the Tempura code of the conveyor as follows (the full program can be found in Appendix B).

```
define conveyor(L, R1, S)={
 always{
   if ((R1 = 0 and L[0] = 0 and L[9] = 0) or
       (R1 = 0 and L[0] = 1 and L[9] = 0) or
       (R1 = 1 and L[0] = 1 and L[9] = 0) )
        then {next S = 0 and next L = [0] + L[0..|L|-1]}
   else{
        if ((R1 = 1 and L[0] = 0 and L[9] = 0) or
            (R1 = 0 and L[0] = 1 and L[9] = 1) or
            (R1 = 1 and L[0] = 1 and L[9] = 1) or
            (R1 = 1 and L[0] = 0 and L[9] = 1) or
            (R1 = 0 and L[0] = 0 and L[9] = 1) )
           then {next S = 1 and
                 next L = (if R1=1 and L[0]=0 then [1]
           else [L[0]])+ L[1..|L|-1] + (if L[9]=1 then [0]
           else [L[9]])
                 }
        }
} /* end of always */
}.  /* end of conveyor */
```

Because this code is executable, we can then run it and check relevant run-time behaviours of the source code in Tempura. A checking facility has been added into the

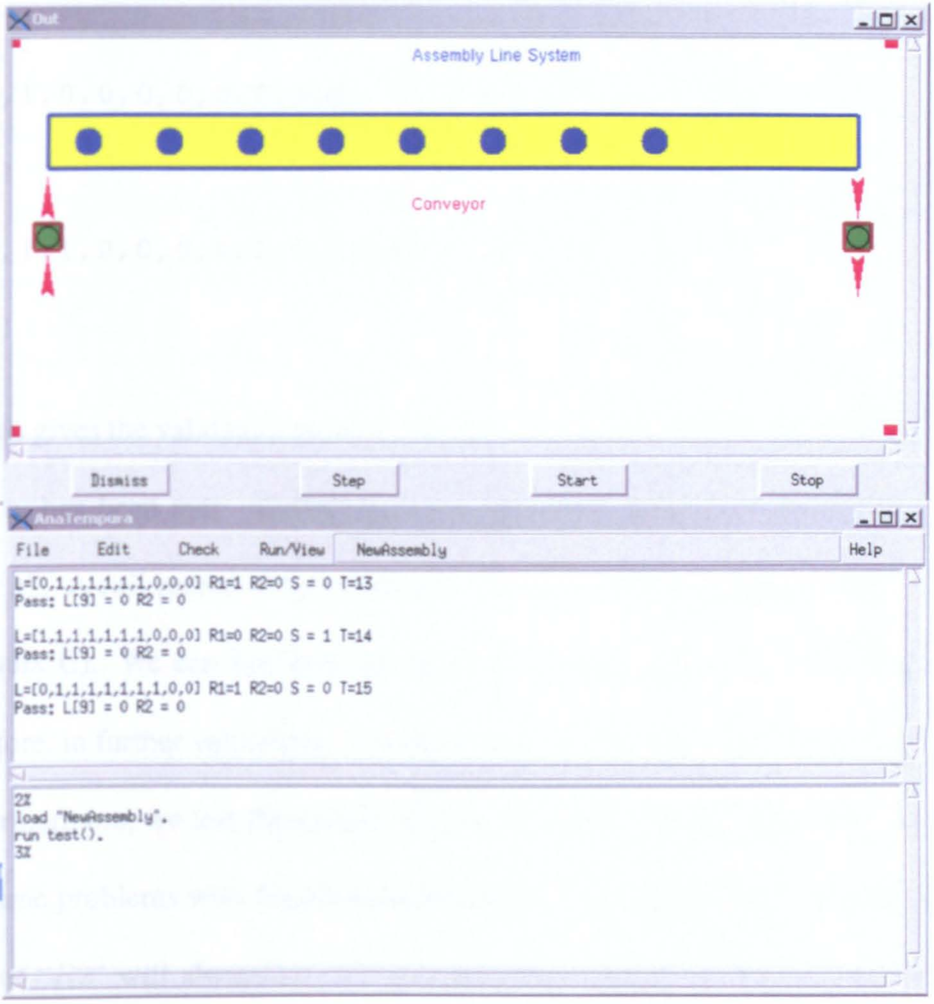
Tempura code directly since accessing to all variables is possible in running the Tempura code. The checking facility will be executed in parallel with the run of the assembly system's source code. We give the checking facility in Tempura as follows:

```
{ if not (L[9] = 1 and R2 = 1)
    then format("Pass! \n\n", L[9], R2 )
    else format ("Fail at %t : L[9] = %t R2 = %t \n\n",
                 L[9], R2)}
```

The checking facility is a simple if-then-else clause. If the run-time behaviour satisfies the safety property, a result of "*Pass*!" will be given. If the run-time behaviour violates the safety property, a result of "*Fail*", together with values of the relevant variables in that state and the time stamp, will be given. We can then analyse it in a later stage. furthermore, if the Tempura is converted into source code in other runnable languages, such as C, with inserting appropriate assertion points, the checking facility can be easily converted to get and check assertion data. For example, we can have a function, "*get*()", to capture values of relevant variables in each state. Then the checking facility will look like:

```
{get(L[9]) and get(R2) and
  if not (L[9] = 1 and R2 = 1)
    then format("Pass! \n\n", L[9], R2 )
    else format ("Fail at %t : L[9] = %t R2 = %t \n\n",
                 L[9], R2)}
```

A screen shot of the validating process is presented in Figure 7.10.



```
L=[0,1,1,1,1,1,1,0,0,0] R1=1 R2=0 S = 0 T=13
Pass: L[9] = 0 R2 = 0

L=[1,1,1,1,1,1,1,0,0,0] R1=0 R2=0 S = 1 T=14
Pass: L[9] = 0 R2 = 0

L=[0,1,1,1,1,1,1,1,0,0] R1=1 R2=0 S = 0 T=15
Pass: L[9] = 0 R2 = 0
```

```
2%
load "NewAssembly".
run test().
3%
```

Figure 7.10: Validation of the Assembly System

Some validating results have been give as follows:

```
L=[0,0,0,0,0,0,0,0,0,0] R1=0 R2=0 S = 1 T=0

Pass!

L=[0,0,0,0,0,0,0,0,0,0] R1=1 R2=0 S = 0 T=1

Pass!

L=[1,0,0,0,0,0,0,0,0,0] R1=0 R2=0 S = 1 T=2

Pass!

L=[0,1,0,0,0,0,0,0,0,0] R1=1 R2=0 S = 0 T=3
```

```
Pass!

L=[1,1,0,0,0,0,0,0,0,0] R1=0 R2=0 S = 1 T=4

Pass!

L=[0,1,1,0,0,0,0,0,0,0] R1=1 R2=0 S = 0 T=5

Pass!
```

This gives the validating process of the first six states. They indicate that the safety property has been met. The assembly system never enters a state of "$L[9] = 1 \wedge R2 = 1$". A full record of the validating process with forty states has been given in Appendix C). We can find that the system becomes stable after the twentieth state. Therefore, in further validation, we only need to test for less than twenty-five rounds.

Furthermore, we test the system with an incorrect case, i.e., supposing the robot 2 gets some problems with transferring workcells. Therefore, in this incorrect case, the value of "$R2$" will always be "1", i.e., the robot 2 holds a workcell but will not be able to release it to a tray. Some test results are given as follows (a full record with twenty-five states can be found in Appendix D):

```
L=[1,1,1,1,1,1,1,1,1,0] R1=0 R2=1 S = 1 T=18

Pass!

L=[0,1,1,1,1,1,1,1,1,1] R1=1 R2=1 S = 0 T=19

Fail: L[9] = 1 R2 = 1

L=[1,1,1,1,1,1,1,1,1,0] R1=0 R2=1 S = 1 T=20

Pass!

L=[0,1,1,1,1,1,1,1,1,1] R1=1 R2=1 S = 0 T=21
```

```
Fail: L[9] = 1 R2 = 1.
```

This means that a workcell will always fall from the right end of the conveyor and the control system begins to fail from the nineteenth state. For all even states after the nineteenth state, because a workcell falls and the right end of the conveyor is vacant, the safety property has been satisfied. However, for all odd states after the nineteenth state, workcells keep falling and being broken, which leads to a general failure of the system.

In this case study, we actually did a general run-time analysis. We do not derive the specification to any specific code, such as C or Java but Tempura executable code. We can still run the code (Tempura code) under a realistic circumstance and test properties of interest. That shows the beauty of Tempura and our approach from another side. We need no assertion points in this general case. We put just the property which we want to check in parallel with the three processes, because we have access to all variables already. Later any programmer can easily convert the Tempura executable code to source code in any other language and insert some simple assertion points. The checking facility inside the Tempura code can be easily modified to check assertion data sent by assertion points.

Phase 3 Deployment

As we mentioned at the beginning of this chapter, because we do not convert the code in any specific language, i.e., we do not produce any final product for this program, we only need to make limited activities in this phase. As outcomes from this phase, we need to include a description to the change requirement, i.e., adding a new robot to the assembly system, all formal specifications, the executable Tempura code, and some typical run-time test and analysis results in the deliverable documentation. A programmer will easily produce a system in other language such as C and Java, as well as producing assertion points and doing further run-time analysis. Main parts of the document have been included in Appendix B, Appendix C, and Appendix D.

Chapter 8

Conclusion

8.1 Summary of Thesis

This thesis describes a study of the problems concerning handling evolutionary development of time-critical systems. A systematic research approach has been optimised to engineer time-critical systems and a tool has been implemented based on this approach.

The investigation was started by addressing the overall process of time-critical systems engineering. Time-critical systems nowadays plays an important role within the broad area of software engineering. Enormous problems of engineering time-critical systems await solutions, especially, problems of handling evolution of time-critical systems. The investigation also shows that there exists little systematic research on engineering time-critical systems. It is well-known that the use of formal methods is fundamental for assuring the correctness of time-critical system. However, there is still a big gap between formal development and run-time evaluation. Therefore, an approach to engineer the evolving time-critical systems systematically is addressed in this thesis.

The approach proposed in this thesis is used to tackle the development and evolution of time-critical systems rapidly, efficiently, and above all correctly. The central stage is to develop an integrated framework to deal with the life-cycle of time-critical systems. This framework integrates conventional approaches and formal technologies for engineering time-critical systems. The basic components of guided evolution of time-critical systems have been identified and defined in the thesis. It provides a technical basis for a repeatable, well-defined, and managed development process. It first addresses a general architectural methodology of handling evolutionary development of time-critical systems. This involves crossing levels of abstraction of time-critical systems, from specification in ITL to source code in different programming languages. Time-critical systems' behaviours of interest can be analysed and validated in any stage of evolutionary development. The validation and analysis are performed within a single logical framework. The assertion points technique has been applied in the thesis to generated run-time data, which fully reflects run-time behaviours of the time-critical systems. The run-time data then will be captured and used to validate behaviours of interest with respect to the formal specification of the system. Errors will be reported during the system run, i.e., the run-time analysis does not only report an error but also indicate the location of the error. The approach can be applied to all systems in any major programming languages, such as C and Java, for which the source code and compiler is available.

Furthermore, compositional theories have been developed and integrated into the framework. A set of extendible compositional rules have been adopted as the main guideline for a repeatable and well-manageable way to handle evolutionary development of time-critical systems.

A prototype tool has been developed to support the proposed approach. The tool is also used to implement the developed guidelines for guided evolution. A number of case studies have been used for experiments with the approach and the prototype system. They demonstrate the success of the proposed approach.

8.2 Criteria for Success and Analysis

In Chapter 1, a set of criteria are proposed to judge the success of the approach described in this thesis. In this section, detailed analysis of our approach are presented based on these criteria.

- *For a "living" time-critical system, what is the most specific characteristic, distinguishing it from other system from the perspective of evolution and making a higher potential for producing impact of change than other conventional systems?*

  The nature of a living time-critical, i.e., highly evolutionary, causes continuous changes. This evolutionary life-cycle of time-critical systems brings even more troubles to developers than other systems because of its specific characteristic, the tight coupling between functional properties and timing properties.

- *Can we have a systematic way to cope with the specific characteristic of time-*

critical systems and its evolutionary life-cycle?

As a prerequisite, we use assumption/commitment style Interval Temporal Logic (ITL) specifications to reveal the relationships between timing and functional properties, as well as impacts between different sub-systems and between the system and its environment. After we can specify the system, its changes and impact of change correctly, we then developed a phase-based methodology, providing a general architectural basis for guided evolution of time-critical systems, crossing different levels of time-critical systems, from specification to source code in different programming languages (language independent).

- *Can timing or functional behaviours of interest of a time-critical system be captured, analysed and validated efficiently and correctly under its real working environment in real-time?*

  By means of ITL, a work-bench of ITL, including its executable sub-set of Tempura, assumption/commitment style specifications in ITL, we can define properties of interest efficiently and correctly. By means of run-time analysing mechanisms, assertion points and a corresponding analysing and validating mechanism, behaviours of interest of a time-critical system can be captured, analysed and validated efficiently and correctly in run time. Assertion points, a mechanism inserted in the system, will generate assertion data. The corresponding analysing and validating mechanism will then capture the assertion data and interpret it into run-time behaviours of the system and then analyse and validate the system via comparing run-time behaviours and properties of interest.

- *Can a time-critical system be developed under a repeatable base?*

  The answer is positive. A compositional framework, i.e., to develop the system's sub-system separately and then to compose the sub-systems together, provides a repeatable base for development and evolution of a time-critical system. That is, by means of composition, if we need to repeat some evolutionary developing process, we only need to work with relevant sub-systems instead of work with the whole, making repeatable development possible. Further, a set of compositional rules can be used to decompose and compose a system so that we can work with sub-systems with manageable size.

- *How easy is it to manage evolutionary development of a time-critical system using the proposed approach?*

  The approach provides an integrated engineering framework for handling evolutionary development of time-critical systems. This involves crossing levels of abstraction of time-critical systems, from specification to source code. The validation and analysis are performed within a single logical framework. The approach defines guided evolution with giving a set of guidelines, including run-time analysis guidelines, compositional guidelines and timing analysis guidelines. All rules were addressed under the integrated framework and are extensible. Visualisation mechanises were developed for revealing different stages of time-critical systems' development. In addition, the systems do not need to be rebuilt from scratch. Only the changes and their ripple effect need to be considered. But this is a minimum that has to be done in every maintenance (evolution) system.

- *Is the approach feasible for realisation? For example, is it possible to build a tool based on the approach?*

  Quite a lot attention was paid to the practical part of the approach during development. The case studies show that the approach is a "practical" one, i.e., feasible for practice. A resulting tool named AnaTempura has been built.

The Tool AnaTempura is designed to support the step-by-step methodology of handling evolutionary development of time-critical systems. This tool helps engineers in handling the evolution of time-critical systems in a comprehensive way.

AnaTempura helps the user by performing its functions in an intelligent way. AnaTempura automatically monitors time-critical system execution and analyses the system's run-time behaviours.

AnaTempura employs visualisation techniques. The tool has a friendly user interface. It provides animation of the program or system's run-time behaviours, as well as draws timing diagrams during run-time. Both animation and timing diagrams are helpful into analysis of the behaviours of the system and reveal the evolutionary development process of the system.

AnaTempura considers possible error cases comprehensively. It is tolerant to many user errors. The tool checks for the errors, corrects the errors whenever possible, and gives relevant prompt information.

In addition, AnaTempura could anticipate user decision and interaction by providing possible operations' prompt and information prompt. AnaTempura can be installed into different platforms, such as UNIX.

Assessment This research has so far indicated that the approach can be used to handle practical time-critical systems. However, the real application of this approach will not be seen until an industrial-strength tool has been developed and deep industrial-based experiments have been carried. Therefore, more supporting units of AnaTempura should be built such that more new features can be implemented and easily added to it. There could be another valuable issus of integrating our approach in existing commercial/industry-strength "evolution" tools.

The research presented in the thesis gives valuable detailed understanding in this field.

The designed approach has been implemented by carrying out a number of case studies. Experiments have been carried out and sample results of these experiments are presented. Therefore, development and implementation of the approach achieved the proposed goal of the research.

The approach developed in the thesis is a successful outcome of the research. It is a valuable contribution to the area of time-critical systems research. This approach combines theoretical development and experimental implementation and thus supplies the confidence required when handling time-critical systems development.

8.3 Future Directions

Based on the discussions in former sections, we concluded that the approach has novel ideas and is useful in handling evolutionary development of time-critical systems. The resulting tool scales up the approach. In addition, our approach can be easily adapted for different types of systems. For instance, with Components of The Shelf (COTS), things we need to do is to produce the assumption/commitment specifications in ITL/Tempura, and then instead of using refinement to develop changed system, we can use COTS. In this section, we explore some possible extensions of the present work.

The most interesting future work could be an automatic assertion points producing and inserting mechanism. In our previous research, we have to produce and insert assertion points manually. It could be a good research issue of intruding some other techniques, such as program understanding, to enable the AnaTempura tool to find appropriate allocations and insert corresponding assertion points.

This research has so far indicated that the approach can be used to handle practical time-critical systems. However, the real application of this approach will not be seen until an industrial-strength tool has been developed and deep industrial-based experiments have been carried. Therefore, more supporting units of AnaTempura should be built such that more new features can be implemented and easily added to it. There could be another valuable issus of integrating our approach in existing commercial or industry-strengthed "evolution" tools.

Furthermore, since all animations were made manually using TK. We have to create a graphic model for each different system or property of interest from scratch. A suitable graphic model could be developed and integrated with the formal ITL model to give the target system more intuitive description so that we can derive an animation system more easily from the ITL graphic model without building things from scratch. Because there are a number of more powerful graphic languages or packages existing, such as VRML for 3D graphics, a future development of AnaTempThis research has so far indicated that the approach can be used to handle practical time-critical systems. However, the real application of this approach will not be seen until an industrial-strength tool has been developed and deep industrial-based experiments have been carried. Therefore, more supporting units of AnaTempura should be built such that more new features can be implemented and easily added to it. There could be another valuable issus of integrating our approach in existing commercial/industry-strength "evolution" tools. ura could be to integrate a suitable external graphic package to enhance the feature of visualisation. A further feature of AnaTempura and the external graphic package could be an automatic matching mechanism. This mechanism could keep the consistency between the ITL specification and the graphic model. Any changes in ITL specifications could be reflected in the graphic model immediately.

Software reuse is an important technique in software development. Reusing possible components in time-critical systems is a valuable research issue. Component-based software engineering covers the study of extracting reusableThis research has so far indicated that the approach can be used to handle practical time-critical systems. However, the real application of this approach will not be seen until an industrial-strength tool has been developed and deep industrial-based experiments have been carried. Therefore, more supporting units of AnaTempura should be built such that more new features can be implemented and easily added to it. There could be another valuable issus of integrating our approach in existing commercial/industry-strength "evolution" tools. components from exiting source code and reusing them in further design and different environments. It could be a good and useful research issue how to apply the our approach to the extraction and validation of reusable components, especially attempting to define various environments which a reusable component could be deployed via using the assumption/commitment technique and the analysing strategy developed in the thesis.

References

1. EUROPEAN SPACE AGENCY. *ESA Software Engineering Standards*, esa pss-05-0 issue 2 ed. rue MarioNikis, 75738 Paris Cedex, France, February 1991.

2. ABADI, M., AND LAMPORT, L. Composing Specifications. Tech. Rep. SRC Report 66, Systems Research Centre, Digital Equipment Corp., Oct 1990.

3. ALUR, R., HENZINGER, T. A., AND HO, P. Automatic Symbolic Verification of Embedded Systems. *IEEE Transactions on Software Engineering 22*, 3 (1996), 181–201.

4. AOYAGI, T., FUJITA, M., AND MOTO-OKA, T. Temporal Logic Programming Language Tokio. In *Logic Programming'85*, E. Wada, Ed., LNCS 221. Springer-Verlag, 1986, pp. 138–147.

5. ARNOLD, R. S. *Software Reengineering*. IEEE Computer Society Press, 1994, ch. Introduction: A Road Map Guide to Software Reengineering Technology, pp. 3–22.

6. ARNOLD, R. S., AND BOHNER, S. A. Impact Analysis - Towards a Framework for Comparison. *Proceedings Int. Conf. Software Maintenance* (1993).

7. ARTHUR, L. J. *Software Evolution: The Software Maintenance Chanllenge*. John Wiley & Sons, 1988.

8. ASSOCIATION, R. I. *Safety Related Software for Railway Signalling*, brb/lu ltd/ria technical specification no. 23 ed. 6 Buckingham Gate, London SW1E 6JP, UK, 1991.

9. BACK, R. J. R. A Calculus of Refinements for Program Derivations. *Acta Informatica 25* (1988), 593–624.

10. BARDEN, R., STEPNEY, S., AND COOPER, D. The use of Z. In *Z User Workshop, York 1991* (1992), J. E. Nicholls, Ed., Workshops in Computing, Springer-Verlag, pp. 99–124.

11. BARRINGER, H., KUIPER, R., AND PNUELI, A. Now You May Compose Temporal Logic Specifications. In *Sixteenth Annual ACM Symposium on Theory of Computing* (1984), ACM, pp. 51–63.

12. BEAR, S. An overview of HP-SL. In *VDM '91, Formal Software Development Methods*, S. Prehn and W. J. Toetenel, Eds., no. 551 in Lecture Notes in Computer Science. Springer-Verlag, 1991, pp. 571–587.

13. BEHFOROOZ, A., AND HUDSON, F. J. *Software Engineering Fundamentals.* Oxford University Press, 1996.

14. BENVENISTE, A., AND HARTER, P. K. Proving real-time properties of programs with temporal logics. In *Proceedings of ACM SIGOPS 8th annual ACM symposium on Operating systems Principles* (Dec. 1981), pp. 1–11.

15. BJØRNER, D., AND JONES, C. B. *Formal Specification and Software Development.* Prentice-Hall International, 1982.

16. BOEHM, B. A Sprial Model for Software Development and Ehancement. *Computer 21*, 5 (May 1988), 61–72.

17. Bohner, S. A., and Arnold, R. S. *Software Change Impact Analysis.* IEEE Computer Society, Los Alamitos, 1996.

18. Bowen, J., and Stavridou, V. Safety-Critical Systems, Formal Methods and Standards. *IEE/BCS Software Engineering Journal 8*, 4 (July 1993), 189–209.

19. British Standards Institute. *VDM Specification Proto-Standard*, March 1991. Draft, BSI IST/5/50 document.

20. Brown, M. H., and Sedgewick, R. A System for Algorithm Animation. *IEEE Computer Graphics 18*, 3 (1984), 177–185.

21. Broy, M. A Functional Rephrasing of the Assumption/Commitment Specification Style. Tech. Rep. TuM-I9417, Technische Univerität München, 1994.

22. Burns, A., and Lister, A. M. A Framework for Building Dependable Systems. *Computer 34*, 2 (1991), 173–181.

23. Burns, A., and Wellings, A. *Real-Time Systems and Programming Languages*, 2nd ed. Addison-Wesley, 1997.

24. Campos, S., and Grumberg, O. Selective Quantitative Analysis and Interval Model Checking: Verifying Different Facets of a System. In *Proceedings of the Eighth International Conference on Computer Aided Verification CAV* (New Brunswick, NJ, USA, July/Aug. 1996), Rajeev Alur and Thomas A. Henzinger, Eds., vol. 1102 of *LNCS*, Springer Verlag, pp. 257–268.

25. Cau, A., and Collette, P. Parallel Composition of Assumption-Commitment Specifications: A Unifying Approach for Shared Variable and Distributed Message Passing Concurrency. *Acta Informatica 33*, 2 (1996), 153–176.

26. CAU, A., AND ZEDAN, H. Refining Interval Temporal Logic Specifications. In *the 4th AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Software (ARTS'97)* (Mallorca, Spain, May 1997).

27. CAU, A., ZEDAN, H., COLEMAN, N., AND MOSZKOWSKI, B. Using ITL and Tempura for Large Scale Specification and Simulation. In *Proceedings of 4th EUROMICRO Workshop on Parallel and Distributed Processing, IEEE* (Braga, Portugal, 1996), pp. 493–500.

28. CHARETTE, R. N. *Application Strategies for Risk Analysis.* Software Engineering Series. McGraw Hill, 1990.

29. CHEN, Z., ZEDAN, H., CAU, A., AND YANG, H. Integrating Structured OO Approaches with Formal Techniques for the Development of Real-time Systems. *Information and Software Technology Journal 41* (1999), pp. 435–450.

30. CLOCKSIN, W., AND MELLISH, C. *Programming in Prolog.* Springer-Verlag, 1981.

31. COLEMAN, D. The technology transfer of formal methods: what's going wrong? In *Proceedings of 12th ICSE Workshop on Industrial Use of Formal Methods* (Nice, France, March 1990).

32. CYRUS, J. L., AND BELDSOE, J. D. Formal Specification and Structured Design in Software Development. *Hewlett-Packard Journal* (December 1991), pp. 51–58.

33. DASDAN, A., AND GUPTA, R. K. Timing Issues in System-Level Design. In *IEEE CS Annual Workshop on VLSI (IWV): System-Level Design* (1998), pp. 124–129.

34. DE ROEVER, W. P. The Quest for Compositionaltiy. In *Proceedings of IFIP Working Conf., The Role of Abstract Models in Computer Science* (1985), Elsevier Science B. V. (North-Holland).

35. DRUSINSKY, D. The Temporal Rover and the ATG Rover. In *Proceedings of SPIN200* (2000), Time-Rover Corp., Springer-Verlag.

36. EMERSON, E. A. Temporal and Modal Logic. In *Hand Book of Theoretical Computer Science*, J. van Leeuwen, Ed. Elsevier, 1990, pp. 996–1072.

37. FENCOTT, C. *Formal Methods for Concurrency.* International Thomson Publishing Company, 1996.

38. FLOYD, R. W. Assigning Meanings to Programs. In *Mathematical Aspects of Computer Science*. American Math. Soc., 1967.

39. FOR AERONAUTICS, R. T. C. Minimum Operational Performance Standards for Traffic Alert and Collision Avoidance System (TCAS) Airborne Equipment - Consolidated Edition. DO185, One McPherson Square, 1425 K Street N.W., Suite 500, Washington DC 20005, USA,, 6 September 1990.

40. GALLAGHER, K. B., AND LYLE, J. Using Program Slicing in Software Maintenance. *IEEE TSE 17* (1991).

41. GENRICH, H. J., AND LAUTENBACH, K. System Modelling with High-Level Petri Nets. *Theoretical Computer Science 13* (1981).

42. GILB, T. *Principles of Software Engineering Management.* Addison-Wesley, Reading MA, 1988.

43. GOGUEN, J. A., AND TARDO, J. J. An introduction to OBJ: A language for writing and testing formal algebraic program specifications. In *Software Specification Techniques*. Addison-Wesley Publishing Company, 1986.

44. GTKWAVE ANALYZER. *http://daggit.pagecreator.com/ver/wave/.*

45. GUTTAG, J., AND HORNING, J. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

46. HALE, R. Temporal Logic Programming. In *Temporal Logics and Their Applications,* A. Galton, Ed. Academic Press, London, 1987, pp. 91–119.

47. HALL, J. A. Seven myths of formal methods. *IEEE Software* (September 1990).

48. HAREL, D. Statecharts: A Visual Formalism for Complex Systems. In *Science of Programming 8* (1986).

49. HARRY, A. *Formal Methods Fact File: VDM and Z.* John Wiley & Sons, 1996.

50. HENZINGER, T. A., HO, P., AND WONGTOI, H. *Lecture Notes in Computer Science 1254.* Springer-Verlag, 1997, ch. HyTech: A Model Checker for Hybrid Systems.

51. HILL, J. V. The development of high reliability software - RR&A's experience for safety critical systems. In *Second IEE/BCS Conference, Software Engineering 88, Conference Publication 290* (July 1988), pp. 169–172.

52. HILL, J. V. Software development methods in practice. In *Proceedings of 6th Annual Conference on Computer Assurance (COMPASS)* (1991).

53. HOARE, C. An axiomatic basis for computer programming. *Communications of ACM 12* (Oct. 1969), 576–580.

54. HOARE, C. Notes on data structuring. In *Structured Programming.* Academic Press, Inc., London, 1972.

55. HOARE, C. Proof of A structured program: The sieve of eratosthenes. *Computer 14*, 4 (1972).

56. HOARE, C. Communicating sequential processes. *Communication of ACM 21* (08 1978), 666–677.

57. HOOMAN, J. Specification and compositional verification of real-time systems. *PhD Thesis* (1991).

58. IEEE. *Software Engineering: IEEE Standards Collection.* Institute of Electrical and Electronics Engineers, Inc., 1997.

59. INC., S. VeriWell User's Guide, 1994.

60. INTERNATIONAL ELECTROTECHNICAL COMMISSION. Software for Computers in the Application of Industrial Safety Related Systems. International Electrotechnical Commission, Technical Committee no. 65, Working Group 9 (WG9), IEC 65A (Secretariat) 122, Version 1.0,, August 1991.

61. INTERNATIONAL ELECTROTECHNICAL COMMISSION. Functional Safety of Programmable Electronic Systems: Generic Aspects. International Electrotechnical Commission, Technical Committee no. 65, Working Group 10 (WG10), IEC 65A (Secretariat) 123, February 1992.

62. INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. *ISO 8807: Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, first ed. Geneva, Switzerland, 15 February 1989.

63. INTERNET. http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?safety-critical

64. JAHANIAN, F., AND GOYAL, A. A Formalism for Monitoring Real-Time Constraints at Run-Time. *20th Int. Symp. on Fault-Tolerant Computing Systems (FTCS-20)* (1990), 148–55.

65. JAHANIAN, F., AND MOK, A. K. Safety Analysis of Timing Properties in Real-Time Systems. *IEEE Transactions on Software Engineering 12*, 9 (1986), 890–904.

66. JAHANIAN, F., AND STUART, D. A. A Method for Verifying Properties of Modechart Specifications. In *Proceedings of Real-Time Systems Symposium* (1988).

67. JONES, C. B. *Development Methods for Computer Programs Including a Notion of Interference*. PhD thesis, Oxford University Computing Laboratory, 1981.

68. JONES, C. B. Specification and Design of (Parallel) Programs. In *Information Processing 83*, R. E. A. Mason, Ed. North Holland, 1983, pp. 321–332.

69. JONES, C. B. Tentative Steps Towards a Development Method for Interfering Programs. *ACM Trans. on Prog. Lang. and Syst. 5*, 4 (1983), 596–619.

70. JOSEPH, M. *Real-Time Systems: Specification, Verification and Analysis*. Prentice Hall, 1996.

71. JOSEPH, M., AND GOSWAMI, A. Formal Description of Real-time Systems: A Review. Tech. Rep. RR129, University of Warwick, Dept of Computer Science, 1988.

72. KOPETZ, H., ZAINLINGER, R., FOHLER, G., KANTZ, H., PUSCHNER, P., AND SCHUETZ, W. An Engineering Approach to Hard Real-Time System Design. In *Proceedings of the 3rd European Software Engineering Conference, ESEC'91* (Milano, Italy, October 1991), pp. 166–188.

73. KOYMANS, R. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*. No. 651 in Lecture Notes in Computer Scinece. Springer-Verlag, 1992.

74. KRAMER, J., MAGEE, J., SLOMAN, M. S., AND LISTER, A. M. CONIC: an Intergrated Approach to Distributed Computer Control Systems. *Proceedings of IEE(Part E) 180*, 1 (1983), 1–10.

75. LADEAU, B. R., AND FREEMAN, C. Using formal specification for product development. *Hewlett-Packard Journal* (December 1991), 62–66.

76. LIBES, D. *Exploring Expect*. O'Reilly UK, 1994. ISBN: 1565920902.

77. LIU, X., YANG, H., AND ZEDAN, H. Formal methods for the re-engineering of computing systems. In *Proceedings of The 21st IEEE International Conference on Computer Software and Application (COMPSAC'97)* (Washington, D. C., August 1997), IEEE Computer Society, pp. 409–141.

78. LOCK, S., AND KOTONYA, G. Abstract: An Integrated Framework for Requirement Change Impact Analysis. *Requirenautics Quarterly: The Newsletter of the Requirements Engineering Specialist Group of the British Computer Society* (2000).

79. LOGRIPPO, L., MELANCHUK, T., AND WORS, R. J. D. The algebraic specification language LOTOS: an industrial experience. *ACM SIGSOFT Software Engineering Notes 15* (April 1990), 59–66.

80. MAHONY, B., AND DONG, J. S. Timed Communicating Object Z. *IEEE Transactions on Software Engineering 26*, 2 (February 2000), 150–177.

81. MAHONY, B. P., AND HAYES, I. J. A Case-study in Timed Refinement: A Mine Pump. *IEEE Trans on Software Engineering 18*, 9 (1992), 817–826.

82. MANNA, Z., AND PNUELI, A. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, ISBN 0-387-97664-7, 1996.

83. MCDERMID, J. Assurance in High Integrity Software. In *High Integrity Software.* Pitman, 1989.

84. MCDERMID, J., AND ROOK, P. Software Development Process Models. In *Software Engineer's Reference Book.* CRC Press, 1993, pp. 15/26–15/28.

85. MCDERMID, J. A. *Safety Aspects of Computer Control'.* Butterworth, 1993, ch. Formal Methods: Use and Relevance of the Development of Safety-critical Systems.

86. MILNER, R. *A Calculus of Communicating Systems.* Springer-Verlag, 1980.

87. MINISTRY OF DEFENCE. Defence Standard 00-56: Safety Management Requirements for Defence Systems, 1996.

88. MINISTRY OF DEFENCE. Defence Standard 00-55: Requirements for Safety Related Software in Defence Equipment, 1997.

89. MISRA, J., AND CHANDY, K. M. Proofs of Networks of Processes. *IEEE Trans on Software Engineering 7*, 7 (1981), 417–426.

90. MOK, A. K. SARTOR- a Design Environment for Real-Time Systems. In *Proceedings of 9th IEEE COMPSAC* (OCT 1985), pp. 174–181.

91. MORGAN, C. *Programming from Specifications.* Prentice-Hall International, 1990.

92. MOSZKOWSKI, B. *Reasoning about Digital Circuits.* PhD thesis, Stanford University, July 1983.

93. MOSZKOWSKI, B. *A Temporal Logic for Multilevel Reasoning about Hardware.* IEEE Computer Society, February 1985.

94. MOSZKOWSKI, B. *Executing Temporal Logic Programs.* Cambridge University Press, Cambridge UK, 1986.

95. MOSZKOWSKI, B. Some Very Compositional Temporal Properties. *Programming Concepts, Methods and Calculi, Ernst-Rudiger Olderog (ed.), IFIP Transactions A*, 56 (1994), 307–326.

96. MOSZKOWSKI, B. Using Temporal Fixpoints to Compositionally Reason about Liveness. In *Proceedings of the 7th BCS FACS Refinement Workshop* (Bath, UK, 1996), J. He, Ed.

97. MOSZKOWSKI, B. A Complete Axiomatization of Interval Temporal Logic with Infinite Time. In *Proceedings of the Fifteenth Annual IEEE Symposium on Logic in Computer Science (LICS 2000)* (Santa Barbara, California, June 2000), IEEE Computer Society Press, pp. 242–251.

98. MOTUS, L., AND RODD, M. G. *Timing Analysis of Real-Time Software.* Pergamon, 1994.

99. MUNSON, J., AND ELBAUM, S. Code churn: A measure for estimating the impact of code change. *Proceedings of IEEE Conference on Software Maintenance'98 (ICSM98)* (1998).

100. NICHOLLS, J. E., AND (EDS.), S. M. B. Z Base Standard. Tech. Rep. ZIP/PRG/92/121, Oxford University Computing Laboratory, ZIP Project, Available from the Secretary, ZIP Project, Oxford University Computing Laboratory, PRG, 11 Keble Road, Oxford OX1 3QD, UK, 30 November 1992.

101. OSTROFF, J. S., AND WONHAM, W. M. A temporal logic approach to real-time control. In *Proceedings of the 24th IEEE Conference on Decision and Control* (Florida, Dec. 1985), pp. 656–657.

102. PAAKKI, J., SALMINEN, A., AND KOSKINEN, J. Automated hypertext support for software maintenance. *The Computer Journal 7* (1996), 577–597.

103. PANGDYá, P. K. Some Comments on the Assumption-Commitment Framework for Compositional Verification of Distributed Programs. In *Stepwise Refinement of Distributed Systems.* , J. W. de Bakker, W. P. de Roever, and G. Rozenberg, Eds., no. 430 in Lecture Notes in Computer Science. Springer, 1990, pp. 622–640.

104. PNUELI, A. In Transition from Global to Modular Temporal Reasoning about Programs. In *Logics and Models of Concurrent Systems*, K. R. Apt, Ed., NATO ASI Series. Springer-Verlag, Berlin, Oct 1984, pp. 123–144.

105. PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach*, 5th ed. McGraw-Hill Book Company, 2000. European Adaptation, adapted by D. Ince.

106. RAJLICH, V. A Model for Change Propagation Based on Graph Rewriting. *Proceedings of ICSM'97* (1997).

107. RAJLICH, V., DAMASKINOS, N., LINOS, P., AND KHORSHID, W. VIFOR: A tool for software maintenance. *Software Practice and Experience 20* (1990).

108. REED, G. M., AND ROSCOE, A. W. Timed CSP: Theory and practice. In *REX Workshop—Real-Time : Theory and Practice* (1992), LNCS Springer-Verlag.

109. REISIG, W. *Petri Nets: an Introduction*. Springer-Verlag, Berlin, 1985.

110. Rescher, N., and Urquhart, A. Temporal logic. *Library of Exact Philosophy* (1971).

111. Rushby, J. Formal specification and verification of a fault-masking and transientrecovery model for digital flight control systems. Tech. Rep. Contractor Report 4384, NASA Langley Research Centre, Hampton, Virginia, USA, 1991.

112. Rushby, J. A Tutorial on Specification and Verification Using PVS. In *Proceedings of the First International Symposium of Formal Methods Europe FME '93: Industrial-Strength Formal Methods* (Odense, Denmark, 1993), P. G. Larsen, Ed., pp. 357–406.

113. Sankar, S., Rosenblum, D., and Neff, R. An Implementation of Anna. In *Proceedings of the Ada International Conference on Ada in Use* (Paris, May 1985), J. G. P. Barnes and J. G. A. Fisher, Eds., ACM, Cambridge University Press, pp. 285–296.

114. Schneider, S., Davies, J., M.Jackson, D., G.M.Reed, Reed, J., and Roscoe, A. Timed CSP: Theory and Practice. In *PRX Real-time Workshop, Real-time: Theory in Practice*, J. de Bakker, C. Huizing, W.-P. de Roever, and G. Rozenberg, Eds., no. 600 in LNCS. Springer-Verlag, 1992.

115. Shoham, Y. *Reasoning about Change. Time and Causation from the Standpoint of Artificial Intelligence*. MIT Press, 1988.

116. Spivey, J. M. *Understanding Z*. Cambridge University Press, 1988.

117. Srivas, M., and Bickford, M. Verification of the FtCayuga fault-tolerant microprocessor system, vol 1: a case study in theorem prover-based verification. Tech. Rep. Contractor Report 4381, NASA Langley Research Centre, Hampton, Virginia, USA, July 1991.

118. STANKOVIC, J., AND RAMAMRITHAM, K. *Hard Real-Time Systems: Tutorial Text.* IEEE Comp. Society Press, 1988.

119. STIRLING, C. A Generalizationof Owicki-Cries's Hoare Logic for a Concurrent While language. *Theoretical Computer Science 58* (1988), 347–359.

120. STØLEN, K. *Development of Parallel Programs on Shared Data Structures.* PhD thesis, Manchester University, 1990.

121. STØLEN, K. A Method for the Development of Totally Correct Shared-state Parallel Programs. In *Proceedings of CONCUR 91, LNCS 527*, J. Baeten and J. F. Groote, Eds. Springer-Verlag, 1991.

122. STØLEN, K. An Attempt to reason about Shared-state Concurrency in the Style of VDM. In *Proceedings of VDM91, LNCS 551*, S. Prehn and W. J. Toetenel, Eds. Springer-Verlag, 1991.

123. STØLEN, K. Development of Parallel Programs on Shared Data Structures. Tech. rep., Manchester University, UNCS 1991-1-1, 1991.

124. STØLEN, K., DEDERICHS, F., AND WEBER, R. Assumption/commitment Rules for Networks of Asynchronously Communicating Agents. Tech. Rep. SFB 342/2/93, Technical University of Munich, 1993.

125. TAKANG, A., AND GRUBB, P. A. *Software Maintenance:Concepts and Practice.* International Thomson Computer Press, 1996.

126. TERWILLIGER, R. B. PLEASE: a Language Combining Imperative and Logic Programming. *SIGPlan Notices 23*, 4 (Apr. 1988), 103–110.

127. Tomas, G., and Ueberhuber, C. W. *Visualization of Scientific Parallel Programs.* No. 771 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

128. Welch, B. *Practical Programming in TCL and TK,* 3rd ed. Prentice Hall, November 1999. ISBN: 0130220280.

129. Wilde, N., and Nejmeh, B. A. Dependency Analysis: An Aid for Software Maintenance. Tech. rep., Software Enginering Research Center, University of Florida, Gainesville, January 1987.

130. Woodcock, J., and Dickinson, B. Using VDM with Rely and Guarantee-conditions. In *Proceedings of VDM'88, The Way Ahead,* LNCS, Vol328. Springer-Verlag, 1988, pp. 434–458.

131. Xu, Q. W. *A Theory of State-based Parallel Programming.* PhD thesis, Oxford University Computing Laboratory, 1992.

132. Xu, Q. W., Cau, A., and Collette, P. On Unifying Assumption-Commitment Style Proof Rules for Concurrency. In *CONCUR'94, LNCS 836* (1994), B. Jonsson and J. Parrow, Eds.

133. Xu, Q. W., de Roever, W., and He, J. The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs. *Formal Aspects of Computing 9,* 2 (1997), 149–174.

134. Xu, Q. W., and He, J. F. A Theory of State-based Parallel Programming: Part 1. In *Proceedings of BCS FACS 4th Refinement Workshop* (Cambridge, 1991), J. Morris and R. Shaw, Eds., Springer-Verlag.

135. YANG, H., LIU, X., AND ZEDAN, H. Tackling the Abstraction Problem for Reverse Engineering in A System Re-engineering Approach. *Proceedings of IEEE Conference on Software Maintenance'98 (ICS98)* (1998).

136. YAU, S. S., NICHOLL, R. A., TSAI, J. J., AND LIU, S. An Integrated Life-cycle Model for Software Maintenance. *IEEE Transactions on Software Engineering 15* (1988).

137. ZHOU, C. C., HOARE, C., AND RAVN, A. P. A calculus of durations. *Information Processing Letters 40* (05 1991), 269–276.

Appendix A

Mail Sorter: Phase 3 Documentation

A.1 Description of the change

The decision was made to add a new functionality of processing Air Mail letters. The new sub-system should still sort both First Class and Second Class letters within the original timing constraints.

Change involves a functional requirement (i.e. adding the new function of sorting Air Mail letters) and a timing requirement (i.e. keep the same performance time as the old system).

A.2 Specifications

The Old Sorter System

$(Ass \supset Com) \sqsubseteq Sort$

$Ass : Letter\_Class = 1st\_Class \vee Letter\_Class = 2nd\_Class$

$Com : (\mathit{fin}\ time - time \leqslant 470 \wedge$

$(Class\_Sensor = 1st\_Class \supset \Diamond(Solenoid3 = ON)) \wedge$

$(Class\_Sensor = 2nd\_Class \supset \Diamond(Solenoid3 = OFF)))$

Timing Information of the Old Sorter System

$SensorName$	$ExecutionTime$	$Deadline$
$ClassSensor$	$Delay1$	$70ms$
$LetterSensor$	$Delay2$	$70ms$
$ActuatorName$	$ExecutionTime$	$Deadline$
$Solenoid3$	$Delay3$	$80ms$
$Solenoid4$	$Delay4$	$250ms$

Timing Property of the Old Sorter System

$$[\mathit{fin}\ time - time = Delay1 \wedge LetterState =$$

$$at\_class\_sensor \wedge \textsf{stable}\,(LetterState)]\ ;\ \textsf{skip};$$

$$[\mathit{fin}\ time - time = Delay4 \wedge LetterState =$$

$$at\_Solenoid\_4 \wedge \textsf{stable}\,(LetterState)]\ ;\ \textsf{skip};$$

$$[\mathit{fin}\ time - time = Delay2 \wedge LetterState =$$

$$at\_letter\_sensor \wedge \textsf{stable}\,(LetterState)]\ ;\ \textsf{skip};$$

$$[\mathit{fin}\ time - time = Delay3 \wedge LetterState =$$

$$at\_Solenoid\_3 \wedge \textsf{stable}\,(LetterState)]$$

The New Component of Sorting Air Mail Letters

$$(Ass_{Air} \supset Com_{Air}) \sqsubseteq Sort_{Air}$$

$$Ass_{Air} : Letter\_Class = Air \vee Letter\_Class =$$

$$1st\_Class \vee Letter\_Class = 2nd\_Class$$

$$Com_{Air} : (\mathit{fin}\ time - time \leqslant X \wedge$$

$$(AirMail\_Sensor = Air \supset \Diamond(Solenoid5 = ON)) \wedge$$

$$(AirMail\_Sensor \neq Air \supset \Diamond(Solenoid5 = OFF)))$$

New Sorter System: Component of Sorting First and Second Class Letters

$\overline{Sort}$:

$(Ass \supset Com) \sqsubseteq \overline{Sort}$

$Ass : Letter\_Class = 1st\_Class \vee Letter\_Class = 2nd\_Class$

$Com : (\mathit{fin}\ time - time \leqslant 470 - X \wedge$

$(Class\_Sensor = 1st\_Class \supset \Diamond(Solenoid3 = ON)) \wedge$

$(Class\_Sensor = 2nd\_Class \supset \Diamond(Solenoid3 = OFF)))$

$$Sys_1 \ \widehat{=}\ Sort_{Air}; \overline{Sort}$$

The Composition Rule

$$\frac{\begin{array}{l}(Ass \supset Com) \sqsubseteq \overline{Sort},\\ (Ass_{Air} \supset Com_{Air}) \sqsubseteq Sort_{Air},\\ Ass_C \supset Ass_{Air},\\ Com_{Air} \supset Ass,\\ Com \supset Com_C\end{array}}{(Ass_C \supset Com_C) \sqsubseteq (Sort_{Air}\ ; \overline{Sort})}$$

The New Sorting System after Composition

$Ass_C : Letter\_Class = Air \vee Letter\_Class =$

$1st\_Class \vee Letter\_Class = 2nd\_Class$

$Com_C : \mathit{fin}\ time - time \leqslant 470 \wedge ($

$((AirMail\_Sensor = Air) \supset \Diamond(Solenoid5 = ON)) \wedge$

$((AirMail\_Sensor \neq Air \wedge Class\_Sensor = 1st\_Class) \supset$

$[\Diamond(Solenoid5 = OFF) \wedge \Diamond(Solenoid3 = ON) \wedge$

$((AirMail\_Sensor \neq Air \wedge Class\_Sensor = 2nd\_Class) \supset$

$\Diamond(Solenoid5 = OFF) \wedge \Diamond(Solenoid3 = OFF)]))$

The New Timing Information

$SensorName$	$ExecutionTime$	$Deadline$
$AirMailSensor$	$Delay5$	$40ms$
$ClassSensor$	$Delay1$	$40ms$
$LetterSensor$	$Delay2$	$50ms$
$LetterSensorB$	$Delay6$	$50ms$
$ActuatorName$	$ExecutionTime$	$Deadline$
$Solenoid6$	$Delay7$	$90ms$
$Solenoid5$	$Delay8$	$50ms$
$Solenoid4$	$Delay4$	$90ms$
$Solenoid3$	$Delay3$	$60ms$

The New Timing Property

Process for sorting Air Mail letters:

$[\mathit{fin}\ time - time = Delay5 \wedge LetterState =$

$at\_air\_mail\_sensor \wedge \mathsf{stable}\ (LetterState)]$; skip;

$[\mathit{fin}\ time - time = Delay7 \wedge LetterState =$

$at\_Solenoid\_6 \wedge \mathsf{stable}\ (LetterState)]$; skip;

$[\mathit{fin}\ time - time = Delay6 \wedge LetterState =$

$at\_letter\_sensor\_B \wedge \mathsf{stable}\ (LetterState)]$; skip;

$[\mathit{fin}\ time - time = Delay8 \wedge LetterState =$

$at\_Solenoid\_5 \wedge \mathsf{stable}\ (LetterState)]$

Process for sorting 1st/2nd Class letters:

$[\mathit{fin}\ time - time = Delay5 \wedge LetterState =$

$at\_air\_mail\_sensor \wedge \textsf{stable}\ (LetterState)]$; skip;

$[\mathit{fin}\ time - time = Delay7 \wedge LetterState =$

$at\_Solenoid\_6 \wedge \textsf{stable}\ (LetterState)]$; skip;

$[\mathit{fin}\ time - time = Delay6 \wedge LetterState =$

$at\_letter\_sensor\_B \wedge \textsf{stable}\ (LetterState)]$; skip;

$[\mathit{fin}\ time - time = Delay8 \wedge LetterState =$

$at\_Solenoid\_5 \wedge \textsf{stable}\ (LetterState)]$

$[\mathit{fin}\ time - time = Delay1 \wedge LetterState =$

$at\_class\_sensor \wedge \textsf{stable}\ (LetterState)]$; skip;

$[\mathit{fin}\ time - time = Delay4 \wedge LetterState =$

$at\_Solenoid\_4 \wedge \textsf{stable}\ (LetterState)]$; skip;

$[\mathit{fin}\ time - time = Delay2 \wedge LetterState =$

$at\_letter\_sensor \wedge \textsf{stable}\ (LetterState)]$; skip;

$[\mathit{fin}\ time - time = Delay3 \wedge LetterState =$

$at\_Solenoid\_3 \wedge \textsf{stable}\ (LetterState)]$

A.3 Mail Sorter: Source Code

```
/* Program bysorternew.c - Extended letter sorter
 * Sorting Air Mail, First Class and Second Class Letters
 */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "assertion.h"
// loading library of assertion points


/* clearing the sensors */
void pulseD5(void)
{
    int wait = 100;
 while (wait--);
 text_out("clearing sensors");

}

/* close solenoid */
> Delivered-
void SolOn(int SolNo)
{
  assertion("solon",SolNo);
}
/* opening solenoid */
void SolOff(int SolNo)
{
    assertion ("soloff",SolNo);
}

/* delay for time units */
void Delay(int time, int dtype)
{
    int counter_1=0;
    int clo;
    char *temp;

    switch (dtype) {
        case 1 :
            temp = "delay1";
            break;
        case 2 :
            temp = "delay2";
            break;
        case 3 :
            temp = "delay3";
            break;
        case 4 :
            temp = "delay4";
            break;
        case 5 :
            temp = "delay5";
            break;
        case 6 :
            temp = "delay6";
            break;
        case 7 :
            temp = "delay7";
            break;
        case 8 :
            temp = "delay8";
            break;
        default:
            break;
    }
    clo=myclock();
    assertion(temp,0);
    while (myclock()-clo<time)
    {
        counter_1++;
    };
    assertion(temp,1);
}
/* sort second and first class letters into hoppers */
/* yellow = first and black = second */
void Sort_to_hoppers(int Dtime1, int Dtime2, int Dtime3,
                     int Dtime4, int Dtime5, int Dtime6,
                     int Dtime7, int Dtime8)
{
    int YellowSet, AirSet, count;
    int class_sensor, letter_sensor, air_sensor;
    YellowSet = 1;
    AirSet = 1;
    SolOn(4);
    SolOn(6);
    for (count = 1; count < 3; count++)
     /* error count, exit on 2 */
      {
      do
        {
```

```
        /* clear colour and letter sensor
           then read letter colour */
        pulseD5();
        scan_sensor ("air_sensor is ?",&air_sensor);
        if (air_sensor == 1 ) {
            assertion("air",1);
            SolOff(4);
            Delay(Dtime1,1);
            SolOn(4);
            Delay(Dtime4,4);
            scan_sensor ("letter sensor is ?",&letter_sensor);
            assertion("lsens",letter_sensor);
            if( ! AirSet)
              {
              Delay(Dtime6,6); SolOff(5);
              Delay(Dtime7,7); AirSet = 1;
              }
        }
        else {
            assertion("air",0);
            SolOff(4);
            Delay(Dtime1,1);
            SolOn(4);
            Delay(Dtime4,4);
            scan_sensor ("letter sensor is ?",&letter_sensor);
            assertion("lsens",letter_sensor);
            if( AirSet)
               {
               Delay(Dtime6,6); SolOn(5);
               Delay(Dtime7,7); AirSet = 0;
               }
            else {
                Delay(Dtime6,6); Delay(Dtime7,7);
            };

            scan_sensor ("class_sensor is ?",&class_sensor);
            if (class_sensor < 2)
               { /* its yellow - activate solenoid 3 */
                assertion("class",1);
                SolOff(6);
                Delay(Dtime5,5);
                SolOn(6);
                Delay(Dtime8,8);
                if( ! YellowSet)
```

```
                    {scan_sensor ("letter sensor is ?",
                                    &letter_sensor);
                     assertion("lsens",letter_sensor);
                     Delay(Dtime2,2); SolOff(3);
                     Delay(Dtime3,3); YellowSet = 1;
                    }
                  }
             else
               { /* its black - deactivate solenoid 3 */
                 assertion("class",2);
                 SolOff(6);
                 Delay(Dtime5,5);
                 SolOn(6);
                 Delay(Dtime8,8);
                 if( YellowSet)
                  {scan_sensor ("letter sensor is ?",
                                    &letter_sensor);
                   assertion("lsens",letter_sensor);
                   Delay(Dtime2,2); SolOn(3);
                   Delay(Dtime3,3); YellowSet = 0;
                  }

               }
         }

         }
       /* repeat until no letter has passed letter sensor  */
       while (letter_sensor != 0);
       }

}



/*-------------------------------------------------------*
 * function main with parameters                         *
 */

int main()
{
    int ch,delay1,delay2,delay3,delay4,
        delay5,delay6,delay7,delay8;
    printf ("Enter key : ?\n");
    scanf ("%d", &ch);
    while (ch>0) {

        printf ("Enter Delay1  : ?\n");
        scanf ("%d", &delay1);
        printf ("Enter Delay2 : ?\n");
        scanf ("%d", &delay2);
        printf ("Enter Delay3 : ?\n");
        scanf ("%d", &delay3);
        printf ("Enter Delay4  : ?\n");
        scanf ("%d", &delay4);
        printf ("Enter Delay5  : ?\n");
        scanf ("%d", &delay5);
        printf ("Enter Delay6 : ?\n");
        scanf ("%d", &delay6);
        printf ("Enter Delay7 : ?\n");
        scanf ("%d", &delay7);
        printf ("Enter Delay8  : ?\n");
        scanf ("%d", &delay8);
        Sort_to_hoppers(delay1,delay2,delay3,delay4,
                        delay5,delay6,delay7,delay8);
        printf ("Enter key : ?\n");
        scanf ("%d", &ch);
    }
    printf ("!PROG: end ::\n");
}
```

A.4 Source Code: assertion.h

```
/** assertion.h
  * library to assertion points
  */

/* get time */
int myclock()
{
    return clock()/1000;
}
/* generate output text */
void text_out(char *txt)
{
    printf("%d:: %s\n",myclock(),txt);
}
/* output a name of sensor */
void scan_sensor(char *txt, int *temp)
{
    text_out(txt);
    scanf("%d",temp);
}
/* output variable name, value and time stamp */
void assertion(char *aname, int val)
{
    printf("!PROG: assert %s:%d:%d:!\n",
                              aname, val, myclock());
}
```

A.5 A Typical Run-time Analysing Result

```
Tempura 1% set path ="/home/staff/sz/ThesisCaseprogram/
Case1Working/CharityVer/Casestudy/Case1/by_sorter".
path was previously set to ""
Tempura 2% load "by_sort1a".
[Reading file /home/staff/sz/ThesisCaseprogram/Case1Working/
CharityVer/Casestudy/Case1/by_sorter/by_sort1a.t]
[Reading file /home/staff/sz/ThesisCaseprogram/Case1Working/
CharityVer/Casestudy/Case1/by_sorter/../library/conversion.t]
[Reading file /home/staff/sz/ThesisCaseprogram/Case1Working/
CharityVer/Casestudy/Case1/by_sorter/../library/exprog.t]
[Reading file /home/staff/sz/ThesisCaseprogram/Case1Working/
CharityVer/Casestudy/Case1/by_sorter/../library/tcl.t]
print_states was previously set to true
Tempura 3% run test().
!0:: Solenoid 4 ON: Pass
!0:: Solenoid 6 ON: Pass
!10:: Air sensor 2: Pass
!10:: Letter sensor B 2: Pass
!10:: Class sensor 0: Pass
!20:: Air sensor 0: Pass
!20:: Solenoid 4 OFF: Pass
!20:: Delay5: Start
!70:: Delay5: End
!70:: Delay5 50: Pass
!70:: Solenoid 4 ON: Pass
!70:: Delay7: Start
!160:: Delay7: End
!160:: Delay7 90: Pass
```

```
!160:: Letter sensor B 0: Pass
!160:: Delay6: Start
!210:: Delay6: End
!210:: Delay6 50: Pass
!210:: Solenoid 5 ON: Pass
!210:: Delay8: Start
!270:: Delay8: End
!270:: Delay8 60: Pass
!270:: Class sensor 2: Pass
!270:: Solenoid 6 OFF: Pass
!270:: Delay1: Start
!320:: Delay1: End
!320:: Delay1 50: Pass
!320:: Solenoid 6 ON: Pass
!320:: Delay4: Start
!370:: Delay4: End
!370:: Delay4 50: Pass
!370:: Delay2: Start
!420:: Delay2: End
!420:: Delay2 50: Pass
!420:: Solenoid 3 ON: Pass
!420:: Delay3: Start
!480:: Delay3: End
!480:: Delay3 60: Pass
!490:: Air sensor 2: Pass
!490:: Letter sensor B 2: Pass
!490:: Class sensor 0: Pass
!500:: Air sensor 1: Pass
!500:: Solenoid 4 OFF: Pass
!500:: Delay5: Start
!550:: Delay5: End
!550:: Delay5 50: Pass
!550:: Solenoid 4 ON: Pass
!550:: Delay7: Start
!640:: Delay7: End
!640:: Delay7 90: Pass
!640:: Letter sensor B 1: Pass
!640:: Delay6: Start
!690:: Delay6: End
!690:: Delay6 50: Pass
!690:: Solenoid 5 OFF: Pass
!690:: Delay8: Start
!750:: Delay8: End
!750:: Delay8 60: Pass
!760:: Air sensor 2: Pass
!760:: Letter sensor B 2: Pass
!760:: Class sensor 0: Pass
!770:: Air sensor 1: Pass
!770:: Solenoid 4 OFF: Pass
!770:: Delay5: Start
!820:: Delay5: End
!820:: Delay5 50: Pass
!820:: Solenoid 4 ON: Pass
!820:: Delay7: Start
!910:: Delay7: End
```

```
!910:: Delay7 90: Pass
!910:: Letter sensor B 0: Pass

Done!  Computation length:  130.  Total Passes:  188.
Total reductions:  49209  (48421 successful).
Maximum reduction depth:  28.
Tempura 4%
```

Appendix B

Assembly System: source code

```
/* Assembly.t--Assembly line with A Conveyor and Two Robots */
/*
 ** three functions for one conveyor and two robots
 ** and then put them together
 ** to check a safety property of
 ** \next \always not (L[9] = 1 \and R2 = 1
 **
*/
load "../library/conversion".
load "../library/tcl".
/* tcl NewAssembly */
set print_states = false.
/*firstly, the code for the conveyor */
/* conveyor move: S= 0  and L = [0] + L[0..|L|-1] */
/* conveyor stop: S= 1  and L = L[0..|L|] */
define conveyor(L, R1, S)={
 always{
   if ((R1 = 0 and L[0] = 0 and L[9] = 0) or
       (R1 = 0 and L[0] = 1 and L[9] = 0) or
       (R1 = 1 and L[0] = 1 and L[9] = 0) )
     then {next S = 0 and next L = [0] + L[0..|L|-1]}
     else {
          if ((R1 = 1 and L[0] = 0 and L[9] = 0) or
             (R1 = 0 and L[0] = 1 and L[9] = 1) or
             (R1 = 1 and L[0] = 1 and L[9] = 1) or
```

```
              (R1 = 1 and L[0] = 0 and L[9] = 1) or
              (R1 = 0 and L[0] = 0 and L[9] = 1) )
             then {next S = 1 and next L =
                  (if R1=1 and L[0]=0 then [1]
                   else [L[0]])+ L[1..|L|-1] +
                      (if L[9]=1 then [0]
                       else [L[9]])
}}}
/* end of always */
}.
/* end of conveyor */
define robotone(L, R1) = {
  always {
   if ((R1 = 1 and L[0] = 0 and L[9] = 0) or
       (R1 = 1 and L[0] = 0 and L[9] = 1) )
     then { next R1 = 0}
     else{
         if ((R1 = 0 and L[0] = 0 and L[9] = 0) or
         (R1 = 0 and L[0] = 1 and L[9] = 0) or
         (R1 = 1 and L[0] = 1 and L[9] = 0) or
         (R1 = 0 and L[0] = 1 and L[9] = 1) or
         (R1 = 1 and L[0] = 1 and L[9] = 1) or
         (R1 = 0 and L[0] = 0 and L[9] = 1) )
          then {next R1 = 1}}
               }/* end of always */
}.
/* end of robot 1 */
define robottwo(L, R2) = {
        always {
                if (R2 = 1)
                   then { next R2 = 0 }
                  else{
                      if (L[9] = 1)
                          then { next R2 = 1 }
                         else {next R2 = 0 }
                      }}/* end of always */
}. /* end of robot 2 */
define assembly(C) = {
      exists L, R1, R2, S, T:{
             tcl("init", [1,2,3]) and
             L=C and T=0 and
             R1=0 and R2=0 and S=1 and format("\n") and
       always{format("L=%t R1=%t R2=%t S = %t T=%t \n",
                     L, R1, R2, S, T )
                and T := T + 1
                and { if not (L[9] = 1 and R2 = 1)
                        then format("Pass!" )
                        else format ("Fail: L[9] =
                                     %t R2 = %t \n\n",
                                    L[9], R2)}
                and always tclbreak()
                   and always tcl("drawconveyone",
                [L[0], L[1], L[2], L[3], L[4],
                 L[5], L[6], L[7], L[8], L[9]] )
                }
                and conveyor(L, R1, S)
                and robotone(L, R1)
                and robottwo(L, R2)}
}.
/* end of assembly */

/* run */ define test() = {
assembly([0,0,0,0,0,0,0,0,0,0])
and len(20)
}.
```

Appendix C

Assembly System: Run-time Analysis Results

```
Tempura 1% set path ="/home/staff/sz/
                       ThesisCaseprogram/Case2Working".
path was previously set to ""
Tempura 2% load "NewAssembly".
[Reading file /home/staff/sz/ThesisCaseprogram/
              Case2Working/NewAssembly.t]
run test().
[Reading file /home/staff/sz/ThesisCaseprogram/Case2Working/
              ../library/conversion.t]
[Reading file /home/staff/sz/ThesisCaseprogram/Case2Working/
              ../library/tcl.t]
print_states was previously set to true
Tempura 3%
L=[0,0,0,0,0,0,0,0,0,0] R1=0 R2=0 S = 1 T=0
Pass!
L=[0,0,0,0,0,0,0,0,0,0] R1=1 R2=0 S = 0 T=1
Pass!
L=[1,0,0,0,0,0,0,0,0,0] R1=0 R2=0 S = 1 T=2
Pass!
L=[0,1,0,0,0,0,0,0,0,0] R1=1 R2=0 S = 0 T=3
Pass!
L=[1,1,0,0,0,0,0,0,0,0] R1=0 R2=0 S = 1 T=4
Pass!
L=[0,1,1,0,0,0,0,0,0,0] R1=1 R2=0 S = 0 T=5
Pass!
L=[1,1,1,0,0,0,0,0,0,0] R1=0 R2=0 S = 1 T=6
Pass!
L=[0,1,1,1,0,0,0,0,0,0] R1=1 R2=0 S = 0 T=7
Pass!
L=[1,1,1,1,0,0,0,0,0,0] R1=0 R2=0 S = 1 T=8
Pass!
```

```
L=[0,1,1,1,1,0,0,0,0,0] R1=1 R2=0 S = 0 T=9
Pass!
L=[1,1,1,1,1,0,0,0,0,0] R1=0 R2=0 S = 1 T=10
Pass!
L=[0,1,1,1,1,1,0,0,0,0] R1=1 R2=0 S = 0 T=11
Pass!
L=[1,1,1,1,1,1,0,0,0,0] R1=0 R2=0 S = 1 T=12
Pass!
L=[0,1,1,1,1,1,1,0,0,0] R1=1 R2=0 S = 0 T=13
Pass!
L=[1,1,1,1,1,1,1,0,0,0] R1=0 R2=0 S = 1 T=14
Pass!
L=[0,1,1,1,1,1,1,1,0,0] R1=1 R2=0 S = 0 T=15
Pass!
L=[1,1,1,1,1,1,1,1,0,0] R1=0 R2=0 S = 1 T=16
Pass!
L=[0,1,1,1,1,1,1,1,1,0] R1=1 R2=0 S = 0 T=17
Pass!
L=[1,1,1,1,1,1,1,1,1,0] R1=0 R2=0 S = 1 T=18
Pass!
L=[0,1,1,1,1,1,1,1,1,1] R1=1 R2=0 S = 0 T=19
Pass!
L=[1,1,1,1,1,1,1,1,1,0] R1=0 R2=1 S = 1 T=20
Pass!
L=[0,1,1,1,1,1,1,1,1,1] R1=1 R2=0 S = 0 T=21
Pass!
L=[1,1,1,1,1,1,1,1,1,0] R1=0 R2=1 S = 1 T=22
Pass!
L=[0,1,1,1,1,1,1,1,1,1] R1=1 R2=0 S = 0 T=23
Pass!
L=[1,1,1,1,1,1,1,1,1,0] R1=0 R2=1 S = 1 T=24
Pass!
L=[0,1,1,1,1,1,1,1,1,1] R1=1 R2=0 S = 0 T=25
Pass!
L=[1,1,1,1,1,1,1,1,1,0] R1=0 R2=1 S = 1 T=26
Pass!
L=[0,1,1,1,1,1,1,1,1,1] R1=1 R2=0 S = 0 T=27
Pass!
L=[1,1,1,1,1,1,1,1,1,0] R1=0 R2=1 S = 1 T=28
Pass!
L=[0,1,1,1,1,1,1,1,1,1] R1=1 R2=0 S = 0 T=29
Pass!
L=[1,1,1,1,1,1,1,1,1,0] R1=0 R2=1 S = 1 T=30
Pass!
L=[0,1,1,1,1,1,1,1,1,1] R1=1 R2=0 S = 0 T=31
Pass!
L=[1,1,1,1,1,1,1,1,1,0] R1=0 R2=1 S = 1 T=32
Pass!
L=[0,1,1,1,1,1,1,1,1,1] R1=1 R2=0 S = 0 T=33
Pass!
L=[1,1,1,1,1,1,1,1,1,0] R1=0 R2=1 S = 1 T=34
Pass!
L=[0,1,1,1,1,1,1,1,1,1] R1=1 R2=0 S = 0 T=35
Pass!
L=[1,1,1,1,1,1,1,1,1,0] R1=0 R2=1 S = 1 T=36
Pass!
L=[0,1,1,1,1,1,1,1,1,1] R1=1 R2=0 S = 0 T=37
Pass!
L=[1,1,1,1,1,1,1,1,1,0] R1=0 R2=1 S = 1 T=38
Pass!
L=[0,1,1,1,1,1,1,1,1,1] R1=1 R2=0 S = 0 T=39
Pass!
L=[1,1,1,1,1,1,1,1,1,0] R1=0 R2=1 S = 1 T=40
Pass!
```

Appendix D

Assembly System: Run-time Analysis Results with A Bad Case

```
Tempura 1% set path ="/home/staff/sz/
                      ThesisCaseprogram/Case2Working".
path was previously set to ""
Tempura 2% load "NewAssemblyBadCase".
[Reading file /home/staff/sz/ThesisCaseprogram/
Case2Working/NewAssemblyBadCase.t]
run test().
[Reading file /home/staff/sz/ThesisCaseprogram/
Case2Working/../library/conversion.t]
[Reading file /home/staff/sz/ThesisCaseprogram/
Case2Working/../library/tcl.t]
print_states was previously set to true
***Syntax Error on line 75 of "/home/staff/sz/
ThesisCaseprogram/Case2Working/NewAssemblyBadCase.t":
"." unexpected.
Tempura 3%
L=[0,0,0,0,0,0,0,0,0,0] R1=0 R2=1 S = 1 T=0
Pass!
L=[0,0,0,0,0,0,0,0,0,0] R1=1 R2=1 S = 0 T=1
Pass!
L=[1,0,0,0,0,0,0,0,0,0] R1=0 R2=1 S = 1 T=2
Pass!
L=[0,1,0,0,0,0,0,0,0,0] R1=1 R2=1 S = 0 T=3
Pass!
L=[1,1,0,0,0,0,0,0,0,0] R1=0 R2=1 S = 1 T=4
Pass!
L=[0,1,1,0,0,0,0,0,0,0] R1=1 R2=1 S = 0 T=5
Pass!
L=[1,1,1,0,0,0,0,0,0,0] R1=0 R2=1 S = 1 T=6
Pass!
L=[0,1,1,1,0,0,0,0,0,0] R1=1 R2=1 S = 0 T=7
```

```
Pass!
L=[1,1,1,1,0,0,0,0,0,0] R1=0 R2=1 S = 1 T=8
Pass!
L=[0,1,1,1,1,0,0,0,0,0] R1=1 R2=1 S = 0 T=9
Pass!
L=[1,1,1,1,1,0,0,0,0,0] R1=0 R2=1 S = 1 T=10
Pass!
L=[0,1,1,1,1,1,0,0,0,0] R1=1 R2=1 S = 0 T=11
Pass!
L=[1,1,1,1,1,1,0,0,0,0] R1=0 R2=1 S = 1 T=12
Pass!
L=[0,1,1,1,1,1,1,0,0,0] R1=1 R2=1 S = 0 T=13
Pass!
L=[1,1,1,1,1,1,1,0,0,0] R1=0 R2=1 S = 1 T=14
Pass!
L=[0,1,1,1,1,1,1,1,0,0] R1=1 R2=1 S = 0 T=15
Pass!
L=[1,1,1,1,1,1,1,1,0,0] R1=0 R2=1 S = 1 T=16
Pass!
L=[0,1,1,1,1,1,1,1,1,0] R1=1 R2=1 S = 0 T=17
Pass!
L=[1,1,1,1,1,1,1,1,1,0] R1=0 R2=1 S = 1 T=18
Pass!
L=[0,1,1,1,1,1,1,1,1,1] R1=1 R2=1 S = 0 T=19
Fail: L[9] = 1 R2 = 1
L=[1,1,1,1,1,1,1,1,1,0] R1=0 R2=1 S = 1 T=20
Pass!
L=[0,1,1,1,1,1,1,1,1,1] R1=1 R2=1 S = 0 T=21
Fail: L[9] = 1 R2 = 1
L=[1,1,1,1,1,1,1,1,1,0] R1=0 R2=1 S = 1 T=22
Pass!
L=[0,1,1,1,1,1,1,1,1,1] R1=1 R2=1 S = 0 T=23
Fail: L[9] = 1 R2 = 1
L=[1,1,1,1,1,1,1,1,1,0] R1=0 R2=1 S = 1 T=24
Pass!
L=[0,1,1,1,1,1,1,1,1,1] R1=1 R2=1 S = 0 T=25
Fail: L[9] = 1 R2 = 1
```