RL-TR-96-57
Final Technical Report
April 1996

SOFTWARE DESIGN FOR REAL-TIME SYSTEMS ON PARALLEL COMPUTERS: FORMAL SPECIFICATIONS

Syracuse University

Alok Choudary, Vijay Geholt, and Bhagirath Narahari

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Rome, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be releasable to the general public, including foreign nations.

RL-TR- 96-57 has been reviewed and is approved for publication.

APPROVED: Milissa M. Benincasa

MILISSA M. BENINCASA
Project Engineer

FOR THE COMMANDER: John A. Graniero

JOHN A. GRANIERO
Chief Scientist
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify Rome Laboratory/ (C3CB), Rome NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 1996	Final Apr 94 - Sep 95

4. TITLE AND SUBTITLE

SOFTWARE DESIGN FOR REAL-TIME SYSTEMS ON PARALLEL COMPUTERS: FORMAL SPECIFICATIONS

5. FUNDING NUMBERS

C - F30602-94-C-0073
PE - 62702F
PR - 5581
TA - 18
WU - PG

6. AUTHOR(S)

Alok Choudhary, Vijay Geholt, and Bhagirath Narahari

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Syracuse University
Department of ECE
Syracuse NY 13244

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Rome Laboratory/C3CB
525 Brooks Rd
Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-96-57

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Milissa M. Benincasa/C3CB/(315) 330-7650

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This research investigated the important issues related to the analysis and design of real-time systems targeted to parallel architectures. In particular, the software specification models for real-time systems on parallel architectures were evaluated. A survey of current formal methods for uniprocessor real-time systems specifications was conducted to determine their extensibility in specifying real-time systems on parallel architectures. In this research, a specification model called Parallel REal Time SpEcification Language (PRETSEL) was defined. It leverages off of existing models while adding the necessary syntax and semantics lacking in existing models in supporting specification of real-time systems for parallel architectures. Examples of utilizing the PRETSEL language are presented.

14. SUBJECT TERMS

Formal methods, Real-time systems, Parallel processing

15. NUMBER OF PAGES

80

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Abstract

Parallel high-performance computing is gaining momentum as a computing platform for many applications including those in science, engineering and command and control. They offer an attractive method to place higher processing requirements, due to more sensors or additional information, on real-time systems. Real-time systems must respond to external events and inputs and exert stimulus on their environment in the form of actuator control, displays, and data and control interaction with other subsystems. Some of the tasks in various C^3 (Command, Control, Communications) applications require processing large number of targets and manipulating extremely large data sets. Future requirements are likely to increase the processing demands due to more sensors and more information, thus suggesting the use of parallel computers to implement real-time systems. In recent years, the research in software support for parallel computers has mainly addressed scientific and information processing applications. Very little attention, if at all, has been paid to real-time embedded system requirements.

This research investigates important issues related to design and analysis of real-time system software for parallel computers. In particular, this paper considers software specification models for real-time systems on parallel computers. A formal specification model will not only allow the designer to specify the system, but will also serve as the basis for automated verification tools that can be used to validate the design. Such tools will allow the design, verification, and validation of complex systems of realistic size. While a number of formal specification languages have been designed for real-time systems, these cannot be expected to adequately model the additional issues introduced by parallelism. This report will conduct a survey of current formal methods for uniprocessor real-time systems, and determine the additional issues introduced by parallelism. The properties and requirements that must be met by a specification model for parallel real-time systems will be defined. The PRETSEL specification model—Parallel REal Time SpEcification Language—is proposed, and its syntax and semantics are developed. PRETSEL extends existing algebraic models by providing structured timing constructs, and explicit parallelism constructs. The

feasibility of building automated verification tools using the PRETSEL language is addressed through the provision of formal operational rules. The PRETSEL approach is then compared with the UNITY approach. Various similarities and differences between the two approaches are identified. On the one hand, the PRETSEL approach is less abstract than the UNITY approach because of the real-time application domain, while on the other hand PRETSEL seems to *conservatively extend* UNITY. The application of PRETSEL in specification of parallel real-time systems is illustrated through various examples.

Contents

1 Introduction

Parallel and distributed computing offers a high speed computing platform for many applications, including those in science, engineering, and command and control. They offer an attractive method to place higher processing requirements, due to more sensors or additional information, on real-time systems.

Real-time systems must respond to external events/inputs and exert stimulus on their environment in form of actuator control, displays, and data/control interaction with other subsystems. Some of the common tasks in various Air force and Navy systems (e.g., E-2C, AWACS, Joint STARS) require processing large number of targets and manipulating extremely large data sets. Future requirements are likely to increase the processing demands due to more sensors and more information, thus suggesting the use of parallel computers to implement real-time systems.

The lack of software support both in the design as well as in the implementation phases has resulted in a slower acceptance of parallel computing than originally expected. Real-time (reactive) systems put even greater requirements on parallel computing software because design criteria must also include "performance", "guarantees of deadlines", "adaptability to external events", and "fault-tolerance". In recent years, the research in software support for parallel computers has mainly addressed scientific and information processing applications. Very little attention, if at all, has been paid to real-time embedded system requirements. The objective of this research project is to investigate important issues related to designing real-time system software for parallel computers.

Complex software systems can be made truly robust and reliable if powerful analysis techniques are made available to software developers and maintainers. Such techniques should be applicable throughout the software life-cycle phases – during development of the system from initial specification, design and coding and the maintenance and modification phases. These techniques should be applicable to a wide range of software system descriptions, program structures and applications, and should be able to analyze systems of realistic size in reasonable time. This requires

that the techniques be automated, and based on sound theoretical models.

The problem of specification, design, and analysis of real-time systems and software is made more difficult by the concurrency in real-time applications and further complicated by the presence of time. Real-time software must satisfy not only functional correctness requirements but also temporal correctness. The imposition of timing constraints reintroduces the dependencies between the software and the hardware capabilities. The fundamental problem in real-time applications is not on the ability to implement them but rather in the costly and often *ad hoc* manner in which they are designed, validated, and maintained. Using testing for system validation is a labor-intensive and error-prone approach and does not lend itself to modifications in the software and to scaling the system to the more complex parallel computer architectures. Consequently, there is a need for automated techniques which will permit a designer to **specify, verify and validate** the real-time system. Mechanization of this process requires a formal theoretical model under which the automation tools can be developed. A formal specification language will allow the system designer to specify the structure of the real-time system and make timing assertions about the system, while leaving the complex problems of resource allocation and verification to automation. Recent projects have produced considerable research on formal specification models and techniques for design and analysis of real-time systems [66], but there has been minimal work on specification of real-time systems implemented on parallel machines. The problem of real-time system specification, and verification, is made more complex when the system is to be implemented on parallel computer architectures.

A real-time reactive system implemented on a parallel computer would consist of many tasks (each of which can be a parallel task) communicating with each other, some handling input data, others processing data and events, some producing output, and each task using the appropriate number of processors for required computation and performance requirements. These tasks will communicate with each other to exchange information and communicate with the external environment for input of data/signal, and to output data/signal. All these tasks must respond appropriately to

any changes in the environment in such a way that the response time and throughput requirements are met according to the specification. However, when parallelism is considered, extrapolation of resources in a straightforward manner to handle the increased parallelism will not work because the scalability of parallel programs depends on many factors, including available parallelism, associated overhead for executing a program on a larger number of processors, and scalability of algorithm itself. These parameters must be adequately specified in the real-time system specification. Therefore, in order to define a real-time system running on a parallel computer, a requirements specification model that supports parallel computing with real-time features is required. Such a high-level specification language must have reliability and verifiability as one of its basic design criteria, and must provide a sound theoretical model upon which semi-automated verification tools can be developed.

1.1 Outline: Goals and Objectives

The objective of this research is to investigate important issues related to designing real-time system software for parallel computers. The specific goal of this research is the investigation of formal methods for specification and semi-automated verification of real-time systems on parallel architectures. To verify real-time processes, the temporal and structural properties of real-time processes must be identified. To describe the behavior of computations on parallel architectures, the identification and specification of the behavior of parallel algorithms based on properties, such as the scalability, communication, efficiency, and the degree of parallelism must be defined. A formal language with rigorous semantics for specifying these properties (of real-time parallel processes), and a formalism for verification of the properties must be defined. These two formalisms provide the basis of automated/mechanized verification tools. Such tools are important since manual verification of a moderate sized program is a time consuming task. This research shall take the approach of investigating current methods for formal specification, and extending and integrating some of the current methods to develop a formal specification model for real-time systems on parallel computers.

This report is organized as follows. First, in Section 2, a survey of the existing formal methods for specification of real-time systems is presented, with particular emphasis on methods that incorporate temporal and concurrency issues. These methods include Temporal Logic, Petri-Nets, Process Algebra, Communicating Sequential Processes (CSP), and PSETL (Parallel SET Language). The survey evaluates the suitability, and drawbacks, of the methods in terms of their applicability to the project. Second, in Section 3, a discussion of the issues introduced by parallelism into the specification problem is presented. The properties and requirements (features) which must be satisfied by a specification model for parallel real-time systems are defined. Thirdly, in Section 4, a computation model for parallel real-time systems upon which a specification model is built, is presented. In Section 5, the syntax of a formal software specification language for real-time systems on parallel computers is presented. This model, called PRETSEL (Parallel REal Time SpEcification Language), extends, and integrates, concepts used in the current models. The PRETSEL specification model must meet the properties, and requirements, demanded by real-time system software on parallel computers. In Section 6, the semantics of PRETSEL and its formal semantic operational rules which thereby provide the basis for automated verification tools that may be developed using the PRETSEL specification model, is presented. In Section 7, the PRETSEL and the UNITY models are compared and contrasted. In Section 8, the effectiveness of the PRETSEL model through examples is illustrated. Finally, Section 9 provides concluding remarks and discusses current and future directions for this project.

2 Formal Specification Models for Real-Time Systems

A formal specification of a software system is a prerequisite for verifying that the program is correct. A formal specification is a precise definition of the logical, and, in the case of real-time systems, temporal properties of the software. A system specification model differs from conventional design specifications in that it is concerned primarily only with the function of the system and makes no commitments to its structure. Writing a formal specification allows software developers to discover errors, clarify and validate requirements, and make decisions about the functionality. It also allows specification while leaving implementation decisions to a later stage. A provision of a formal specification model allows the system designers to make assertions about the specification itself and, more importantly, make assertions about the correctness (functional and temporal) of the programs. Design of such automated tools for verifying and validating the system correctness will allow the development of large realistic real-time systems.

2.1 Specification of Real-Time Systems: Requirements

A specification language for real-time processes must express hard timing constraints and the possible structures of real-time processes. In addition, it should be abstract enough to represent top-level (prescriptive) as well as implementation-level (descriptive) specifications. It should have rigorously defined semantics that reflect the execution of real-time processes. Furthermore, it must address a number of different types of timing constraints that are placed on real-time systems. Some of these types of timing constraints [35, 5, 40, 11, 46, 30] are summarized as follows:

- Event b must not occur later than τ seconds after event a
- The process must wait at least τ seconds after event a occurs before engaging in event b.
- Events a and b are separated by exactly τ seconds

- Action a requires τ seconds to complete
- If event a does not occur within τ seconds of the start of process P, P will time out.
- Process P begins executing every τ seconds
- Process P may be activated at any time, but consecutive activations must be separated by at least τ seconds.

The term *event* is used to mean an action that marks an instant in time, and the term *action* refers to some computation in which a real-time process engages. Thus an action may have a duration while an event may not. In terms of parallel architectures, both events and actions may be parallel algorithms and therefore both may have a duration.

The first five cases express timing constraints in terms of events and actions that occur during the execution of a process. The last two cases express the constraints in terms of process execution rather than specific events. The first two cases specify upper and lower bounds on the intervals between events. The third case gives an exact time by which an event must occur. The fourth case specifies the duration of an action. These are also known as the minimum, maximum, and durational timing constraints. The fifth example is a *timeout* – a process is subject to a timeout if it must execute some event by time n. If the event is not executed by this time, then either the process fails or another process is invoked to handle the timeout. The sixth case is an example of a *periodic process.* That is, a process that begins executing every τ time units, starting at time 0. The final constraint is an example of a *sporadic process* – these may begin executing at any time. The requirement that consecutive executions be separated by τ time units prevents potentially infinite executions within finite time.

2.2 Survey of Formal Specification Models

A number of promising paradigms for formal specification and verification of real-time systems, in which each computation is executed on a sequential computer, have been proposed and studied. These include CSP, CCS, temporal logic, process algebra, Petri-nets, and very high level languages (such as PSETL, and Z). Some of these formalisms were originally designed for specification and verification of concurrent processes, but since have been augmented to include timing specification which allow them to be used for real-time processes.

Depending on the formalism, one can use it either descriptively or prescriptively or both. The former means to give the details of an actual system such as the number of subprocesses involved, their respective behaviors and the ways in which the subprocesses interact. The latter means specifying the desired behavior of a system without specifying how that behavior is to be obtained. Temporal logic has been use prescriptively, CSP, process algebra, Petri nets, and PSETL (and like) can be used both descriptively and prescriptively. In this regard, PSETL (and like) has the advantage in that it is a real programming language from which an actual implementation can be easily derived. This is usually done by compiling such a language to an application language.

The next gives a brief description of some of the existing methods for specification and verification of real-time systems.

Temporal Logic

Temporal logic is a modal logic that allows one to reason about the truth of statements over time. Temporal logic has been used for specification and verification of program properties. Special temporal operators, such as $\Box$ (every) and $\Diamond$ (within), are introduced for analysis of temporal connectives in languages. These temporal operators have been found to be useful for specifying program behavior. The structure of states (such as a sequence or tree of states) is the key concept that makes temporal logic suitable for program specification [56, 58]. In a programming language, the structures

represent the computations executed by a program, and such a computation may be used to interpret a temporal formula.

To use temporal logic for verification, axiomatic semantics for a programming language are defined using the logic. Programs are then verified using the axioms of temporal logic. Extensions to temporal logic have been proposed for specification and verification or real-time properties [53, 1]. Two such extensions are Real Time Temporal Logic (RTTL) and Metric Temporal Logic (MTL).

RTTL (Real Time Temporal Logic): This extension assumes existence of a global clock and has been studied extensively with respect to its application to a number of different real-time applications[53, 54, 55, 56]. A distinguished variable t, called the *clock variable*, is used to refer to clock ticks. Predicates involving this variable then constitute timing specification. For example,

$$t = 2$$

is true in a state when the clock has ticked twice, and

$$w_1 \wedge (t = T) \rightarrow \Diamond(w_2 \wedge (t \leq T + 5))$$

states that "once w_1 is true, w_2 must become true no more than 5 ticks later."

The TTM/RTTL (Timed transition models/Real time temporal logic) framework was first introduced in [56]. It has a semantic model of time, a generic computational model (timed transition model) for modelling plants and controllers, an abstract specification language (RTTL), and verification methodologies including model-checking for finite state systems and a deductive proof system for infinite state systems. It also provides heuristics for constructing proofs and controller synthesis.

MTL (Metric Temporal Logic) [1]: This extension includes a *time bounded* version of the usual temporal operators. Thus,

$$\Box(p \rightarrow \Diamond_{\leq 3} q)$$

means "every event p is followed by q within 3 time units." References to an explicit clock are not allowed, and hence MTL is a hidden clock (or bounded) temporal operator logic. An important extension to MTL is a compositional proof system for

OCCAM style programs [28] in which the proof system uses the maximal parallelism mode of program execution. The proof system is compositional, thus allowing properties of a compound system to be deduced from its constituent parts. This property is important since it allows scaling up the application of the proof system to deal with a large system in a structured fashion [52].

Communicating Sequential Processes (CSP)

Communicating Sequential Processes (CSP) [27] provides a structured method for analysis of discrete event concurrent systems. The provision of a few constructs in CSP lead to a language capable of expressing parallel and distributed computations. The constructs include sequential and parallel composition, nondeterministic choice, and recursion. Based on this several computational models have been developed, and some of these also lead to methods for compositional verification. Algebraic laws relating the constructs allow for transformation of one system into another. In CSP, if P is a process and a an event then $a \rightarrow P$ denotes a process that first engages in event a and then behaves exactly as the process P [27]. Shared events require participation of both processes involved. An example of which is communication over a channel in which a message is sent by one process and received by another. Most methods for proof methods in CSP are based on bottom-up approaches and utilize a proof system, wherein the verification task is reduced to a tautology checking of statements written in the language [52]..

Timed CSP is an extension of Communicating Sequential Processes (CSP) [27, 12, 13]. In the timed extension of CSP, the prefix operator of CSP is decorated with a time value. Thus,

$$a \xrightarrow{t} P$$

represents a process that is willing to do action a. If a occurs, it will behave as P once a delay of time t has elapsed. Timed CSP assumes the existence of a global clock and that the occurrence of an event has zero duration!

Calculi of Communicating systems (CCS)

Calculi of Communicating systems (CCS) is an algebraic formalism closely resembling CSP [44, 45]. In CCS a system is verified by using the notion of a bisimulation, where a system has a specification and an implementation. The axioms of the algebra can be applied to prove equality between specification and implementation[52].

Some models, such as timed CSP based models, assume synchrony which results in some interesting temporal properties of processes being inexpressible, or they assume a global clock and require actions to occur at precise moments of that global clock which subsequently has its drawbacks. In [67] the authors introduce the notion of timing into the CCS model. However, in their model processes could only evolve simultaneously via communication while time and actions were interleaved. The Temporal Calculus of Communicating Systems, proposed in [47], extends the model in [67] by allowing time to pass independent of the functional aspects of a process. The process state transition system is split into two orthogonal parts, one describes the functional aspects of the process and the other describes its temporal aspects. This allows for the separation of functional and temporal concerns in analysing process behaviour. However, this model assume that actions have no duration though they could model within their language actions with duration by requiring a process to take some amount of time in stabilising into a new state.

In [24] the authors propose the Calculus of Communicating Shared Resources which has an underlying resource based computational model and a syntax that closely resembles the CCS syntax [43]. This model allows for explicit modelling of resources and priorities of actions. It provides operators for timeout and interruption. Other timed extensions to CCS include [68] and [69].

Process Algebra

A process algebra consists of an algebraic language that is a collection of function symbols or combinators, and a semantic interpretation of this language [26, 25]. The semantics gives rise to a set of equations that can be used as a proof system for

the algebra. The algebraic language can be used as a specification language, and the verification can be performed by using an equational proof system [32]. The algebraic paradigm provides a single paradigm for specifying and verifying real-time processes. It can be used descriptively to give the details of an actual system such as the number of subprocesses involved, their respective behaviors and the ways in which the subprocesses interact.

Timed Process Algebra extends the standard process algebra model by including a distinguished action σ among its set of action [46, 30]. This action represents a "clock tick." Thus, in this approach, a clock itself is assumed a process generating ticks. Thus,

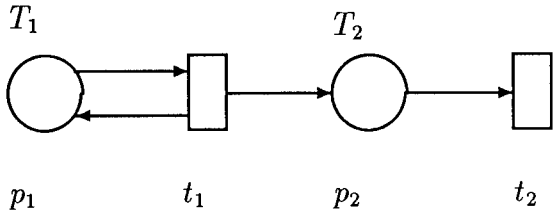
$$m.(\overline{a}.A + \sigma.m.\overline{a}.A)$$

represents a process that sends a message m and if it does not receive an acknowledgment a (receiving is indicated by a bar over the corresponding action) within next clock tick (i.e., one time unit), it resends the message and waits for an acknowledgment.

A number of formal models based on timed process algebra have been developed. These include Real Time ACP [2], Algebra of Timed Processes[48, 49, 50, 51], Urgent LOTOS [7], and Process Algebra for resource bound systems [36]. Although these extensions account for time, they are not designed explicitly to handle timing constraints. Furthermore, they do not provide constructs for specifying the additional parameters introduced by parallel architectures.

Petri Nets

Extensive work has been done in using Petri-Nets to model concurrent and real-time systems. Timed Petri Nets are again extensions of standard Petri Nets which have been used in the modelling of control flow in asynchronous parallel systems [10, 22]. In the timed version of Petri Nets, a time duration is associated with a place and/or transition. The former can be viewed as processes, and the latter as events. Thus, given the timing constraint that $T_1 \geq T_2$, the following timed Petri Net



represents a periodic process with period 1 (assuming $T_1 = 2$, $T_2 = 1$). Petri Nets lack structural constructs and operators. This renders them not amenable to modular/decompositional approach to specification and verification. Hence they are less suited to specification and verification of large and complex systems.

PSETL and SETL

The language SETL belongs to the class of "very high level languages." It is based on general finite sets and maps [62]. Most programming and specification languages have been modeled around an existing mathematical theory, e.g., algebra, mathematical logic, lambda calculus, or relational calculus. Set theory is yet another system that includes most of scientific reasoning, i.e., most scientific facts can be expressed in the language of set theory and shown to be true or false using its methods. In this sense, the language SETL is based on sets just as the language LISP is based on lambda calculus.

SETL is well suited for *specification* and *rapid prototyping* because it is built on data structures that are powerful aggregates, and thus allows one to say much using very few statements. Being a very high level language, SETL also meets, to a good extent, the Department of Defense's (DOD) requirements for a Common Prototyping System (CPS) and Common Prototyping Language (CPL) [20]. The prototyping capabilities of SETL were demostrated by the validated Ada compiler written in SETL[15]. PSETL [29] extends the SETL language to handle parallelism. However, it only considers data parallel programs thereby limiting its scope in specifying asynchronous parallel and distributed systems.

A number of other specification languages have been designed, such as Z [17, 19] and UNITY[9], which do not currently deal with real-time issues. A detailed evaluation of UNITY against the proposed language (PRETSEL) is discussed in a later section.

2.2.1 Some Limitations of Existing Models/Methods

The formal methods surveyed closely model conventional real-time systems consisting of interacting sequential tasks. However, these methods lack in several respects with regard to the application domain defined for this effort.

- Most do not permit specification of different timing constraints (such as periodic, sporadic, within, etc.)
- Most are geared towards specifying only one aspect of system requirements, namely, the timing requirements. They do not integrate, say, functional requirements with the timing requirements.
- All assume a single global clock with no error, which is an idealization. In a real application, there may be more than one clock, each being imperfect.
- None allows for the specification of performance/scheduling requirement.
- None of the methods address implementation on parallel computers which introduces additional complexities, which are discussed in greater detail in the next section.

The formal methods proposed and developed in the real-time literature surveyed are seen to closely model many real-time systems. However, since they were not designed for the explicit purpose of implementing real-time systems on parallel computers, they do not adequately address issues raised by parallelism into the specification problem. Clearly, a specification model for real-time systems on parallel computers must overcome all of the above shortcomings in addition to tackling the issues introduced by parallelism. Some of these issues are briefly discussed in the next section.

3 Specification of Parallel Real-Time Systems

The problem of specifying real-time systems is further complicated when these systems are considered for implementation on parallel computer architectures. These architectures introduce a number of additional issues, and new parameters, which must be considered by a specification model. In the next subsection, these issues are presented. Also a discussion of the desirable properties for a specification model and the specification requirements (and features) placed on such a model for parallel real-time system is described. Finally, the last subsection discusses which of the current formal methods are best suited for extending in order to specify parallel real-time systems.

3.1 Issues Introduced by Parallel Processing

In real-time systems, performance correctness (i.e., meeting deadlines etc.) is as important as functional correctness. However, performance on parallel computers, now also depends on a number of architectural and algorithmic properties such as the number of processors, communication, scalability of algorithms, overhead of scheduling parallelism, and synchronization. These additional characteristics and issues which are introduced by parallel computing must be adequately specifiable in the formal specification model. Some of the issues introduced by using parallel computing, which are normally absent from uniprocessor systems, include:

- Non-deterministic behavior, non-repeatable execution and pure parallelism: This occurs due to variations in interleaving of concurrent activities, non-deterministic language constructs, and asynchrony of external events. The important issue here is the analysis of non-determinism. For example, can perturbation be used to force different orderings of executions to validate designs?

- In real-time systems, performance correctness (i.e., meeting deadlines etc.) is as important as functional correctness. Performance, however, now also depends

on architectural and algorithmic features such as communication, scalability of algorithms, overhead of scheduling parallelism and synchronization. These characteristics must be verified and validated as the part of the design.

- Most reactive systems have to meet real-time requirements that cover a fairly broad spectrum. On one end of the spectrum are hard-real-time control applications where several periodic processes must each meet their deadlines. On the other end are some C3 applications, where fast or soft-real-time processing is required. Therefore, scheduling and mapping policies must incorporate peculiarities of parallel systems. For example, in traditional software for parallel computers, overheads of scheduling such as time to execute scheduling algorithms, synchronization costs, time to load and switch tasks are normally ignored. These cannot be ignored when designing real-time systems. Furthermore, adapting and re-mapping for utilizing greater parallelism in a set of tasks in order to respond to external events and bursty I/O must be specifiable and verifiable.

- One of the major advantages of a parallel computer system is the additional available processing power for specific critical tasks, when necessary. This is especially the case in the presence of bursty IO. This occurs when a system may suddenly encounter a large amount of data to be processed. In such cases, re-mapping strategies can be used to allocate more processors to the critical tasks to meet the performance requirements. However, decisions must be made on how to re-map and reschedule, and the performance must be verifiable at design time. Constructs and directives are needed, which can be used to specify scheduling and re-mapping policies parameterized by bursty IO and time dependent inputs.

- Polyperformance metrics allow performance to be defined using a number of metrics and require specification of multiple versions for a computation.

Why Performance Polymorphism

The performance of a parallel algorithm depends on a number of factors such as degree of parallelism, the data characteristics and data size, and the system characteristics (such as the number and type of processors and the communication channels). Typically the speedup per processor, also called the efficiency, of a parallel algorithm decreases with the increasing number of processors (due to more communication). This scalability parameter must be included in the program specification. Different parallel algorithms for the same computation can have different efficiency functions, where the efficiency depends on factors such as the data size and the number of processors. Consider two different algorithms for the same problem with different performance characteristics: Algorithm 1 and Algorithm 2. For example, in the problem of sorting, insertion sort and heap sort could be the two different algorithms. The performance demanded by the system, to meet the real-time constraints, can be defined by a user-defined metric called *equiperf*. Figure 1, shows the number of processors required, for each data size, by each algorithm to meet the equiperf requirements. As the size of the data varies, the type of algorithm to use, to meet the equiperf requirements, may vary. For example, in Figure 1, when the data size is larger than n Algorithm 1 needs more processors to meet the equiperf requirements while Algorithm 2 needs more processors when the data size is smaller than n. Thus, if the data size is larger than n, Algorithm 2 is a better choice, than Algorithm 1.

The above discussion presents the need for multiple functions/algorithms to carry out a given computation. The specification model to be defined must specify these multiple versions. This concept is called a *performance polymorphism* [31]. The specification must model the performance metrics as a function of the data size, system size, degree of parallelism and other factors. Depending on the state of the system, the properties of the data, and the performance requirement, the appropriate algorithm is selected. The process of selection of the algorithm and the subsequent resource allocation process are part of the resource allocation system and form a critical component of research in developing scheduling and other system support for

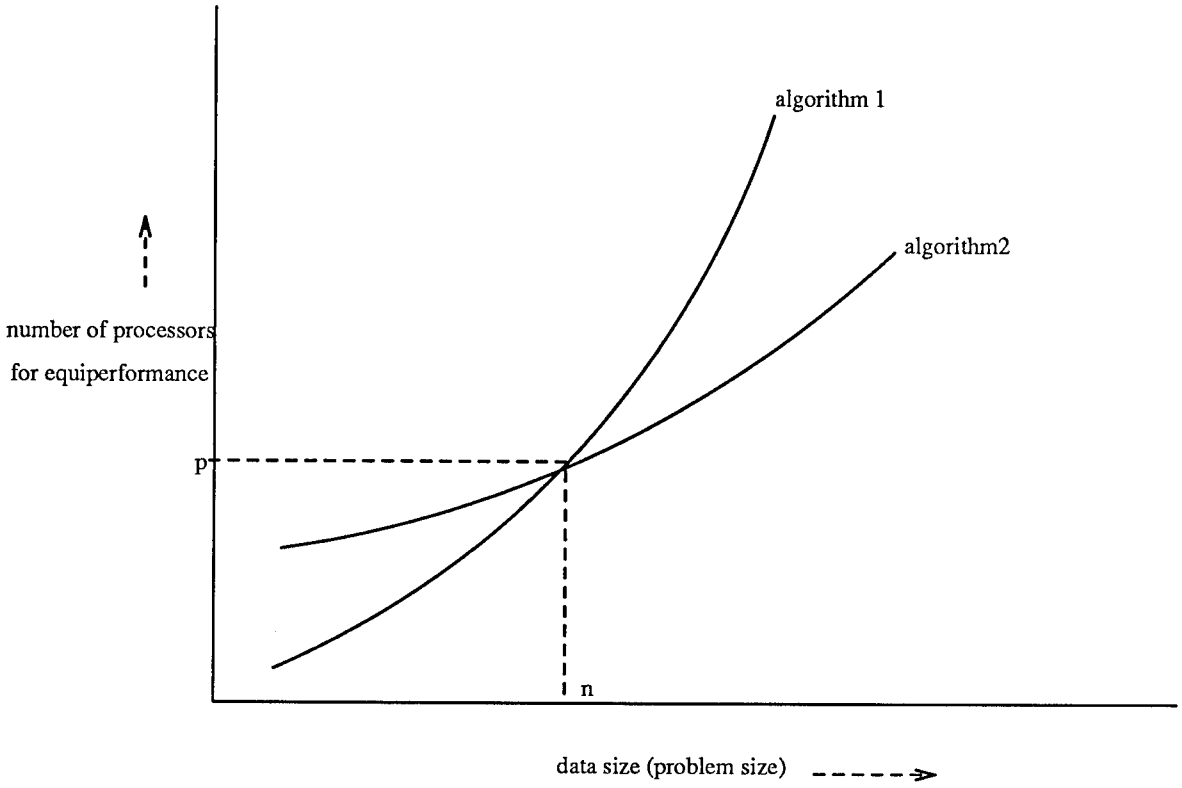


Figure 1: Performance Polymorphism

parallel real-time systems.

3.2 Properties of Specification Model for Parallel Real-Time Systems

In order to define a parallel real-time application a requirements specification notation that supports both parallel and distributed computing with real-time features is required. Also, since most real-time applications tend to be critical, such a high-level notation must have reliability and verifiability as one of its basic design criteria. This formalism must lead to semi-automated verification tools.

It is widely believed (in linguistics theory) that the structure of a specification notation defines boundaries of thoughts. This hypothesis also holds for programming languages where we talk of computational processes instead of thoughts. Over the past decade or so, it has been realized that the good old approach of writing real-time programs in conventional languages (including assembly language) is inadequate in terms of **expressivity**, **portability**, **reliability**, and **verifiability**. This has led to research effort in two directions. The first research direction is design and development of languages that support real-time features. These include PEARL[40], Esterel[5], Real-time Euclid[65], Flex[31], etc. However, none of these language meet the criteria for a real-time language for distributed and parallel computing. Furthermore, with the exception of Esterel, all other real-time languages have largely ignored the issue of verification/validation of programs written in such a language. The second direction is development of various formal models for real-time computing. These include Timed CSP[27], CSP-R[33], Timed Process Algebra[26], etc.. These formal models provide a framework for an automated verification tool. However, although these models give a formal basis for real-time computing, none of these can be construed as a real-time language to write programs in. Therefore, an ideal specification model must operate under a formal basis while providing features that allow the user to write their real-time programs in this specification language.

In light of the discussion above, and the importance of parallel and distributed real-time computing, an investigation into a requirements specification model suitable

for defining real-time applications is required. This model must operate under a formal model while providing sufficient features to write the real-time programs using this notation. Such a model should strive to satisfy the following desirable properties:

- Provide constructs for defining a variety of **time constraints**. Constructs for **timed communication**, constructs for specifying **scalability parameters**. A preliminary effort in this direction has been the real-time extension of the Distributed Programming System (DPS) [35].
- The specification language should be capable of defining a real-time system at the requirements and design phases of the software life-cycle.
- Must be sufficiently high-level so as to facilitate **readability** and **writability**.
- Should be based on the principles of **orthogonality** and **simplicity**.
- Should be useful for describing **synchronous/asynchronous** and **parallel/distributed** computation.
- The model must be given a well-defined formal semantics so as to facilitate **verification/validation** and **correct implementation**.
- Must have facilities for ensuring **reliability**.
- Must allow applicability of existing meta-linguistic real-time formalisms for specification and verification of programs written in the model. These include RTL[30], Temporal logic[59], and Hoare logic with time[64].

In this report, the PRETSEL language—A Parallel REal Time SpEcification Language—for specification of real-time systems on parallel machines is proposed to satisfy the above properties.

3.3 Specification Requirements

The specific requirements that must be satisfied by the specification model for parallel real-time systems, and outline a list of features and constructs that must be provided by the model is now presented. There are three types of constructs that must be provided by a specification requirements model for real-time parallel processes: (1) constructs for timing requirements, (2) constructs for parallelism requirements, and (3) constructs for functional requirements. Current models, for specification of sequential or distributed real-time systems, provide constructs for functional requirements and provide primitive low-level timing specification. Therefore an effective specification model for parallel real-time systems must:

- Have the ability to define structured timing requirements.
- Define the performance requirements and system specification. This should be capable of recognizing changes in system or input parameters (such as change in I/O rate).
- Specify the scalability of the parallel algorithm. This must include performance as a function of the number of processors.
- Provide explicit synchronization constructs - for example, ability to specify timed barrier synchronization, which may itself require specification of partial orders.
- Specify structured communication primitives. These may include point-to-point, permutations, one-to-many, and many-to-one communication patterns.
- Pure Parallelism constructs for process creation and for composition of primitives.
- Provide a capability to specify multiple versions for a computation – each with different characteristics. This is required by the concept of performance polymorphism.

- Address resource allocation such as the mapping of processes to processors
- Predictability of performance - where performance can be approximately derived (lower/upper bounds) from the timing specification.

3.4 Extending Current Formal Models

The potential advantages and disadvantages of some of the current formal models, if they are to serve as the underlying formal basis upon which a new specification language is to be built, is presented below. The aim of the literature survey was to critically evaluate these formalisms with the objective of selecting the formalism best suited for specifying real-time parallel processes. The different formalisms have been grouped under two classes: the algebraic approaches (which include CSP, CCS, and Process Algebra models), and the other models (such as Petri-nets and temporal logic). As a result of the survey, the view taken is that Petri-nets and temporal logic are not suitable candidates upon which to build a specification model, while CSP, CCS and Process Algebra share some common advantages. In particular:

- Disadvantages of using the Petri-Net model:
  - It is non-compositional. The property of composition is important if the specification of a large complex process (such as a parallel program) is to be defined as a composition of simpler processes.
  - It has a low-level syntax.
  - It lacks modifiability, in the sense that an incremental modification of the specified process requires redefining the entire petri-net.
  - It lacks an abstraction mechanism.
- Disadvantage of using the Temporal Logic model: It is a prescriptive model and not a descriptive model. In other words, it does not lend itself to easy implementation from the specification since, low-level details are not specified.
- Advantages common to CSP/CCS/Process Algebra:

- Algebraic specification.
- They are compositional.
- Clear abstraction mechanisms.
- Provide extendibility.
- They are descriptive models.
- There is the possibility of deriving executable specifications.
- LOTOS - international standard based on CCS and CSP.
- Semi-automated verifiers are available (eg. concurrency workbench), which provide the potential of exploring verification methods.

These conclusions have motivated the consideration of using an algebraic model such as CCS as the underlying formalism upon which the specification language is to be built. The CCS model has a number of advantages with respect to meeting the defined requirements. These include algebraic framework, compositional, it provides clear abstraction mechanisms, and semi-automated verifiers are available which provide the potential of exploring verification methods.

4 A Computation Model for Parallel Real-Time Systems

A computation model for real-time systems implemented on parallel machines, which underlies the PRETSEL language, is now defined.

A parallel computation is, in general, a collection of interacting tasks, each of which is a parallel algorithm. At this stage, each task is defined to be a data parallel algorithm, *i.e.*, a real-time parallel computation is a set of interacting data parallel algorithms. (Note that a sequential algorithm can be defined as a data parallel algorithm on a single processor.) Modelling parallel computations in this manner naturally leads to a *two-level* specification model. At level 1, constructs are provided for specifying data parallel algorithms, and at level 2, constructs are provided to combine such tasks in a variety of ways. Thus, parallelism occurs at two levels—within a task (data parallelism) and among tasks (functional or task parallelism).

A data parallel algorithm consists of three activity phases: (1) input and distribution of data, including a external synchronization step, (2) compute-communicate cycles, and (3) output of data and external synchronization. The distribution of data across the processors, and the time taken by the algorithm is a function of the number of processors and the size of the data. These (number of processors and data size) factors themselves can be specified as part of the algorithm. It is noted that the compute-communicate cycle is a synchronous activity. The down-loading of the code, and data, forms one phase of the algorithm. This process, the *load* phase, is performed by the system and entails a system call. In general this time is not taken into account when the programs have been loaded statically. However, if an algorithm is dynamically invoked then this load time must be accounted for. These issues will be addressed later in the report.

In terms of a system implementation, for a data parallel process described above, the assumption is made that there is a table stored in the system. This table has a number of entries, where each entry will have a number of fields/parameters which include: number of processors, data size, input rate, executable code (or pointers to

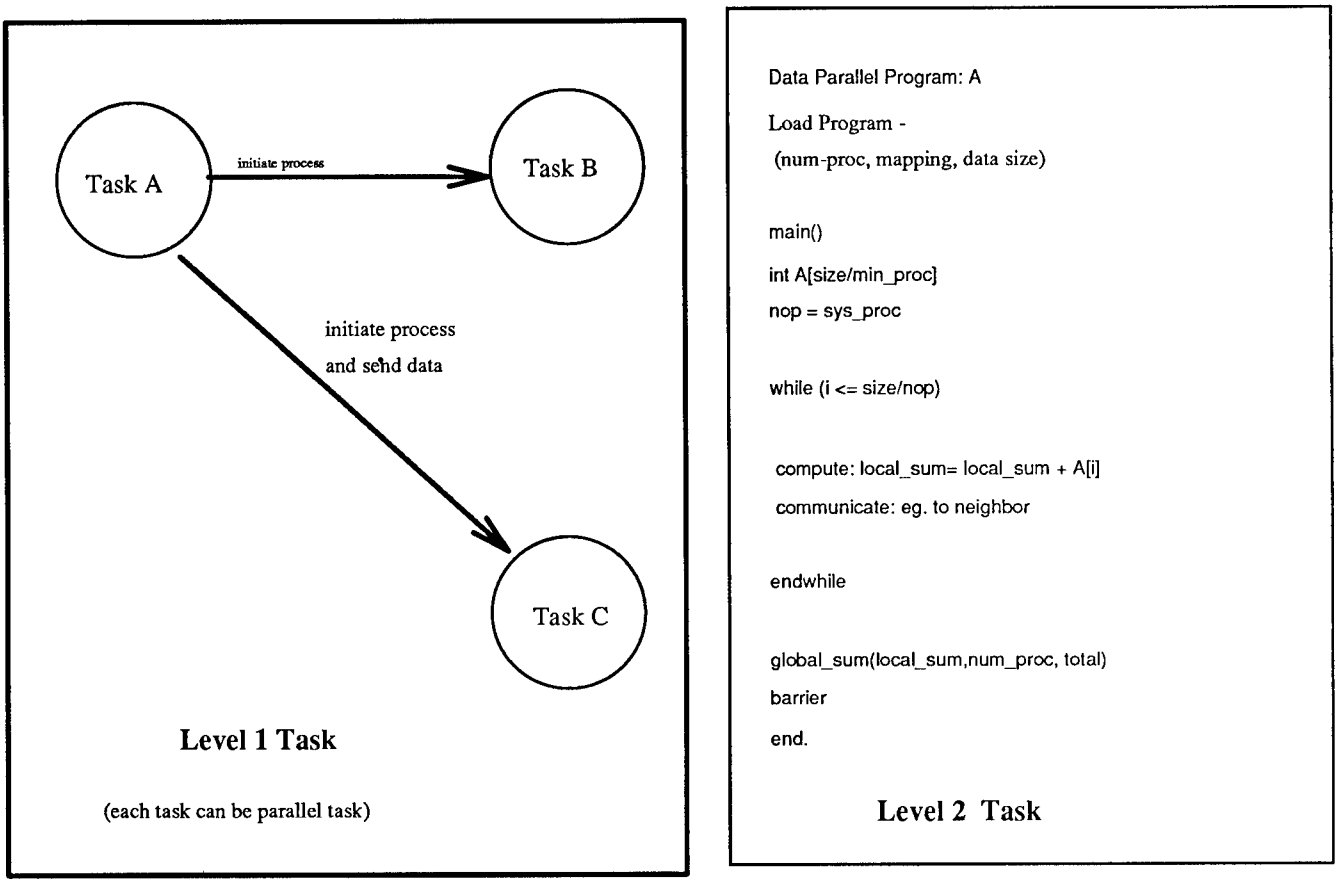


Figure 2: Computation Model

number of processors	data size and distribution	data/input rate	execution time	exec. code

Figure 3: Table of Algorithms

it) and its load time, and the time taken to execute the code. Under this model, it is possible to have different algorithms stored for the same problem, *i.e.,* the executable code stored in one of the fields can vary depending on the values of the other fields.

The data parallel tasks can be combined, to get a level 2 process, using a number of operators which reflect different conditions and dependencies among such data parallel tasks. For example, two data parallel tasks may be executed concurrently (pure parallel composition) or may be executed sequentially (a sequential composition) due to some data dependence.

In a number of applications, the results (data) computed by one parallel task must be sent to another parallel task. This communication of data between two parallel tasks at level 2 is done using a *parallel-send.* After data is computed by task 1, it does a parallel-send to task 2, *i.e.,* task 2 gets its next data set. This scheme is needed, and useful, for periodic tasks. The global period, for this parallel-send operation, may be

computed as the least time to communicate among all parallel tasks at level 2, which can be defined to be the least common divisor of all the periods.

There are two types of communications incurred by tasks at level 1: internal and external. The internal communications are those required by the data parallel algorithm and could include send-receive instructions, permutations, many-to-one, one-to-many, many-to-many, and global reduction operations. Each of these would incur different overheads. This implies that the specification must include the type of internal communication in order to correctly derive the temporal properties of the system. The external communication is that required between tasks at level 2 (between a level 1 task and another level 1 task); for example the parallel-send construct discussed previously would constitute an external communication. These communication primitives will be required for process initiation, when a level 2 process may require initiation, and for synchronization.

Tasks at level 2 represent a parallel (data-parallel) algorithm at level 1, and can be combined to form a real-time process. Thus, a parallel real-time process is defined as interacting tasks at level 2. Since each of the constituent subtasks at level 2 could be different, the modelling of functional parallelism is allowed. There can be any form of precedence between tasks at level 2, and the entire precedence is defined by the level 2 process. The time taken by each subtask at level 2 depends on the time taken by the data parallel algorithm; thus, derivation (or verification) of the time at level 2 requires derivation (or verification) of time taken at level 1.

To be able to model time, the assumption that all actions are recorded with reference to a global clock is made. This obviates the need to model clock synchronization at the specification level.

5 Syntax of PRETSEL

The PRETSEL specification language is based on the computation model described in the previous section. Thus PRETSEL syntax is divided into level 1 syntax and level 2 syntax. The latter provides various constructs to describe a data-parallel algorithm whereas the former contains operator to combine such tasks in a variety of ways. A PRETSEL specification therefore consists of a level 1 process which is a combination of level 2 tasks.

It is worthwhile to point out that one of the design goals of PRETSEL has been that it be usable by even a non-expert. To this end, PRETSEL provides familiar programming language like constructs to define a data-parallel task at level 2. Furthermore, at the present stage of design, PRETSEL does not support recursion as it makes it hard to obtain reasonable time bounds.

To define PRETSEL, a set of action symbols Act is stipulated. The time domain $\mathcal{T}$ is the set of natural numbers plus infinity, that is, $\mathcal{T} = \mathcal{N} \cup \{\infty\}$. Since all actions consume time, it would be convenient to think of an action as a tuple $(\langle label \rangle, \langle time\_spec \rangle)$ where the first component denotes the name of the action and the second component describes its timing specification (described below). Furthermore, assume two mappings $\lambda : Act \rightarrow String$ and $\delta : Act \rightarrow \mathcal{T}$ to extract the name and the timing constraint of an action. For example, if an action $a = (a, \Omega t)$ then $\lambda(a) = a$ and $\delta(a) = t$. Also assume that Act is partitioned into Act_c for pure computation actions, Act_i for internal (*i.e.* level 2) communication actions, Act_e for external communication actions, and Act_s for special actions. Also assume that Act_i and Act_e can be partitioned into two equinumerous sets with a complementation bijection, denoted $\bar{\cdot}$, between them satisfying $\bar{\bar{a}} = a$. Note that a and $\bar{a}$ must have the same timing constraint. The set of PRETSEL level 1 processes $Proc$ is given by the grammar in Figure 4 where min_time and max_time range over the time domain $\mathcal{T}$. The syntax of level 2 tasks is shown in Figure 5.

Let P, T, and t, possibly subscripted, range over the process expressions at level 1, task expressions at level 2, and time domain, respectively. The informal meaning of

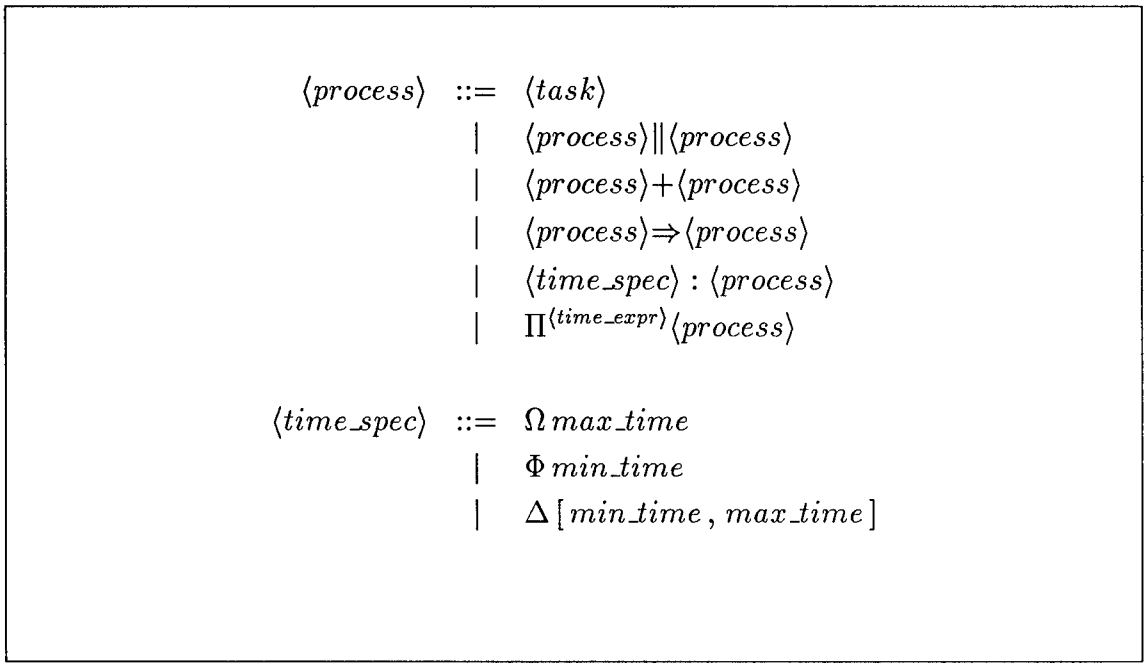


Figure 4: Level 1 Syntax

various operators at level 1 is as follows. The parallel composition $P_1 \| P_2$ denotes a process where two components P_1 and P_2 proceed in time independently of each other except for synchronization. Only the external communication actions may participate in these synchronizations. The sequential composition $P_1 \Rightarrow P_2$ denotes a process where the initiation of the second component P_2 takes place only after the successful termination of the first component P_1. The choice operator $+$ in the expression $P_1 + P_2$ allows the computation to proceed according to either P_1 or P_2, however if P_1 can finish before P_2 then P_1 is selected and vice versa. In this way the choice operator allows for specifying different versions of an algorithm to perform the same computation, such that the algorithm that meets the deadlines will be selected.

Currently, there are three types of timing constraints (or specifications): (1) $\Phi\, min\_time$, (2) $\Omega\, max\_time$, and (3) $\Delta\,[t_1, t_2]$. The first specifies the minimum time, i.e., lower bound requirement, for the computation. The second specifies the maximum time, *i.e.,* upper bound, for the computation. The third specifies a dura-

tion interval, between t_1 and t_2, for the computation. Note that exact timing can be defined using the duration operator as in $\Delta[t, t]$. This specifies that the computation time be exactly t. A process may optionally be *explicitly timed* using the timing constraint operators as in $\Omega t : P$. This expression is only meaningful, if P is *time correct* (this will become clear with the presentation of temporal rules in the next section). The periodic operator Π can be used to define a periodic process at level 1.

Now consider level 2 syntax which specifies data parallel algorithms. At this level a task may be abstracted (or parameterized) by the system specification. This will allow, for example, scalability parameters to be captured by the model. The system specification can include system specific information such as the architecture characteristics (number, type and speed of processors), input characteristics (size and type of data), the mapping function to illustrate how data is distributed across the processors, and the execution time characteristics which can be the execution time as a function of the scalability parameters. At level 2 the basic unit of computation is an *action*. Actions may be combined in several ways to form a composite action or a task. To model real-time behavior a timing constraint is associated with each action. For example, $(\mathtt{add}, \Omega 2)$ describes a basic action that takes a maximum of two units of time to complete. As mentioned above, basic actions can be categorized as pure computations, pure internal communication (communication within the algorithm), external communication (for synchronization) and, in addition, some special actions such as termination and τ action. The first three form the three phases of data parallel algorithms defined by the model of computation. The computations can be arithmetic operations. The internal communications includes: 1) blocking send (*send*) and blocking receive (*receive*) which are used for *synchronous communication*, since the parallel composition at level 2 is essentially synchronous and 2) *barrier* which is essentially a join operation and allows for pure synchronization, that is, no value is exchanged. The various versions of the send and receive operations above, can be used to define the external communications. This includes: 1) parallel send (*par_send*) and parallel receive (*par_receive*) which are used for asynchronous communication at level 1, and 2) pure synchronization operation (*global_sync*). A synchornous send-

receive operation has not been provided at this level because the parallel composition at level 1 is essentially asynchronous and also because the data that is transmitted between level 1 processes would usually be large and thus best suited for aynchronous transfer. One can, however, simulate the effect of synchronous transfer using the asynchronous primitives and *global_sync* operation. The various versions of send and receive operations above can be used to define *many-to-one*, *one-to-many* and *multicast* operations in both blocking and nonblocking mode. An example of this is given in Figure 6.

The basic actions can be combined in parallel using the synchronous parallel operator & or in sequence using the ; operator. The *if* operator allows a deterministic choice to be made based on the boolean expression. The *while* operator allows iterative computations. The time taken by a while operator is derived from the length of the iterations. The *within* operator defines a temporal scope which is meaningful if its body is *time correct.* The *every* operator is used to define a periodic task at level 2. These operators have been adopted from [35]. It should be noted that the operator & is similar to the binary case of **forall** of [8]. Since such **forall**s are so pervasive in parallel programming, the following derived operator is defined:

$$\&(n)T \stackrel{\text{def}}{=} \underbrace{T \,\&\, T \,\&\, \ldots \,\&\, T}_{n}$$

where n is intended to range over the number of processors.

PRETSEL also supports a variety of communication and synchronization mechanisms.

$$
\begin{array}{rcl}
\langle task \rangle & ::= & \langle basic\_task \rangle \mid [\langle sys\_specs \rangle] \langle basic\_task \rangle \\
\\
\langle sys\_specs \rangle & ::= & \langle sys\_spec \rangle, \langle sys\_specs \rangle \\
\\
\langle sys\_spec \rangle & ::= & num\_proc \mid input\_spec \mid exec\_time\_spec \\
& \mid & map\_spec \mid arch\_spec \\
\\
\langle basic\_task \rangle & ::= & \langle action \rangle \\
& \mid & \langle basic\_task \rangle \,\&\langle basic\_task \rangle \\
& \mid & \langle basic\_task \rangle \,;\langle basic\_task \rangle \\
& \mid & if\ \langle bool\_expr \rangle \langle basic\_task \rangle \langle basic\_task \rangle \\
& \mid & while\ \langle bool\_expr \rangle \langle basic\_task \rangle \\
& \mid & every\ \langle time\_expr \rangle \langle basic\_task \rangle \\
& \mid & within\ \langle time\_expr \rangle \langle basic\_task \rangle \\
\\
\langle action \rangle & ::= & \langle comp\_event \rangle \\
& \mid & \langle icomm\_event \rangle \\
& \mid & \langle ext\_comm\_event \rangle \\
& \mid & \langle special\_event \rangle \\
\\
\langle icomm\_event \rangle & ::= & send(\langle expr \rangle) \\
& \mid & receive(\langle var \rangle) \\
& \mid & barrier \\
\\
\langle ext\_comm\_event \rangle & ::= & par\_send(\langle expr \rangle) \\
& \mid & par\_receive(\langle var \rangle) \\
& \mid & global\_synch
\end{array}
$$

Figure 5: Level 2 Syntax

$$blocking\_multi\_send \;:=\; while\ i <= n\ (send(v_i);\ incr\ i)$$

$$blocking\_multi\_receive \;:=\; while\ i <= n\ (receive(x_i);\ incr\ i)$$

Figure 6: Simulating blocking multi-send and multi-recive

6 Semantics of PRETSEL

The above discussion provided an informal view of the semantics of PRETSEL, and now a discussion of the operational semantic rules for PRETSEL is presented. The operational meaning of PRETSEL operators may depend on *temporal correctness* of processes and tasks. To capture temporal correctness, a set of *temporal rules* is defined. For sake of brevity and simplicity, the restriction to Ω constraints are made here. Figure 7 and Figure 8 give the temporal rules for level 1 and level 2, respectively. Both the operational rules and the temporal rules are presented in a natural deduction style. These rules are to be read as follows: if the transition(s) above the line can be inferred, then the transition below the line can be inferred. A special case is when there is nothing above the line. In this case, the transition below the line can be inferred unconditionally. Such rules are also called *axioms*.

$$\frac{P_1 : t_1 \qquad P_2 : t_2}{P_1 \| P_2 : max(t_1, t_2)} \quad (1)$$

$$\frac{P_1 : t_1 \qquad P_2 : t_2}{P_1 + P_2 : min(t_1, t_2)} \quad (2)$$

$$\frac{P_1 : t_1 \qquad P_2 : t_2}{P_1 \Rightarrow P_2 : (t_1 + t_2)} \quad (3)$$

$$\frac{P : t \qquad t \leq t'}{(\Omega t' : P) : t'} \quad (4)$$

$$\frac{P : t \qquad t \leq t'}{\Pi^{t'} P : \infty} \quad (5)$$

Figure 7: Level 1 Temporal Rules

The temporal rules define a relation between the processes and time domain, that is, $: \subseteq Proc \times \mathcal{T}$. The temporal semantics are then defined by the least such relation. Just as typing rules in a typed language assign meaningful types to objects in the

$$\frac{}{a : \delta(a)} \tag{6}$$

$$\frac{T_1 : t_1 \qquad T_2 : t_2}{T_1 ; T_2 : t_1 + t_2} \tag{7}$$

$$\frac{T_1 : t \qquad T_2 : t}{T_1 \,\&\, T_2 : t} \tag{8}$$

$$\frac{T_1 : t_1 \qquad T_2 : t_2 \qquad b : t_3}{if\ b\ T_1\ T_2 : max(t_1 + t_3, t_2 + t_3)} \tag{9}$$

$$\exists n \in \mathcal{N} \cup \{\infty\} \frac{T : t \qquad b : t_1}{while\ b\ T : (n \times (t + t_1)) + t_1} \tag{10}$$

$$\frac{T : t_1 \qquad t_1 \leq t}{every\ t\ T : \infty} \tag{11}$$

$$\frac{T :< t_1 \qquad t_1 \leq t}{within\ t\ T : t} \tag{12}$$

$$\forall \vec{v} \in Val_{sys} \frac{T[\vec{v}/\vec{x}] : t}{[\vec{x}]T : t} \tag{13}$$

Figure 8: Level 2 Temporal Rules

language, the temporal rules may be thought of as assigning temporal information to expressions. In the case where the restriction to Ω is made, the semantics associates the maximum execution time to each process expression. In addition, these rules also provide the temporal meaning to the various operators as follows. According to rule (1), a parallel composite of two processes $P_1 \| P_2$ completes when both its components have completed and hence the time taken is the maximum of the time taken by either component. Rule (2) states that the choice composite of two processes $P_1 + P_2$ finishes as soon as one of them is done. Rule (3) states that for sequential composition $P_1 \Rightarrow P_2$ the maximum time requirement to complete is the sum of the times required by its components. According to rule (4), a process may be constrained by a time operator only if the corresponding value is time compatible with the execution time of the component process. Rule (5) states that the execution of a periodic task may not be bounded and that the period must be compatible with the execution time requirement of the body process. Rule (6) is an axiom. Rule (7) is analogous to rule (3) for processes. Rule (8) captures the synchronous nature of the components of & operator. Thus, it requires that both T_1 and T_2 in $T_1 \,\&\, T_2$ have the same timing behavior. Rule (9) states that the time to complete an *if* operation is the maximum of the time taken to complete the consequent and the alternative. According to rule (10), the maximum time taken by a *while* construct depends on number of iterations. Rule (11) is analogous to rule (5) for processes. Rule (12) states that the temporal scope of a task must be compatible with the timing requirement of its body. Rule (13) requires some explanation. Assume the existence of a value space Val_{sys} for all the system related parameters. In practice this space would be finite and could be maintained as a lookup table. The rule states that the timing requirement of a parameterized task is nothing but the timing requirements inferred after substituting values for each of the system parameters in the task abstraction. Thus, an abstracted task represents a collection of timing requirements. This allows multiple versions of an algorithm to be defined each possibly having a different performance characteristic.

The aforementioned temporal rules can be used to either verify or infer useful temporal information. As a small example, consider a simple process that does the

add operation and then sends a signal. Thus $P = \texttt{add};\texttt{send}$. Further suppose that on a given machine it is known how long the add operation is going to take, say, $\delta(\texttt{add}) = 2$, but it is not known how long the send operation takes. Furthermore, suppose that P is to finish in 10 time units, that is, $\Omega 10 : P$ is what is needed. Using rules(4), (6), and (7) it can be deduced that the send operation must be completed within 8 units of time. This is depicted in the proof tree below:

$$\dfrac{\text{rule 7}\ \dfrac{\text{rule 6}\ \dfrac{}{\texttt{add} : 2} \qquad \text{rule 6}\ \dfrac{}{\texttt{send} : x}}{(\texttt{add};\ \texttt{send}) : y} \qquad y \leq 10}{(\Omega 10 : \texttt{add};\texttt{send}) : 10}\ \text{rule 4}$$

The desired deduction follows in trying to build (backwards) a proof-tree of the goal $(\Omega 10 : \texttt{add};\texttt{send}) : 10$. From the application of rule 4, it can be deduced that the desired goal is provable if we can establish that $(\texttt{add};\texttt{send}) : y$ and $y \leq 10$ for some y. From rule 7, it can be deduced that this y must be $2 + x$, where x is the unknown timing requirement for the send operation. From the constraint $y \leq 10$, it is immediately deduced that $x \leq 8$. Thus, this kind of information can be statically deduced and can be used at compile time for scheduling etc.

Next, the focus is on operational rules. The operational rules for level 1 and level 2 are contained in Figure 9 and Figure 10, respectively. The operational rules are transition based. In defining these rules, a is allowed to range over Act, i range over Act_i, and e range over Act_e. Also, the special action *done* is only present in the semantic domain, that is, it cannot be used to construct process expressions. It is used to flag the termination of a process activity. The operational rules define a relation $\rightarrow \subseteq Proc \times Act \times Proc$. The operational semantics are then defined by the least such relation. The notation $P \xrightarrow{a} P'$ means that the process P behaves like process P' after doing action a and in doing so, it consumes $\delta(a)$ time. Thus operational rules allow us to record what actions a process can perform and how much time it takes. It should be noted that since there is no separation between time and action, it is not necessary to define two separate transition relations as has been done in [47]; rather the approach presented is similar to that of [36], though differs from it in that the

timing requirements of an action are explicit instead of being implicit.

The operational rules give meaning to the various operators as follows. According to rules (14) and (15), a sequential composition $P_1 \Rightarrow P_2$ can only engage in the actions of P_1 as long as it is not finished. It can only start to engage in actions of P_2 after P_1 has terminated. Rules (16) and (17) define the meaning of the choice operator. Thus in $P_1 + P_2$ if P_1 can finish first then according to rule (16) P_1 will get selected. If, however, P_2 can beat P_1 then rule (17) applies and P_2 gets selected. In case both have exactly the same requirements, the choice becomes nondeterministic. Rules (18)-(20) assign meaning to the parallel operator $\|$. According to rule (18), if in the composite $P_1 \| P_2$, the process P_1 is ready to engage in an action and P_2 is not ready to engage in the complementary action, then the only action possible for the composite is that of P_1. Similarly, according to rule (19), if P_2 is ready to engage in an action and P_1 is not ready to engage in the complementary action, then the only action possible for the composite is that of P_2. However, if P_1 and P_2 are ready to engage in complementary actions, they must synchronize. This is the essence of rule (20) and this is what is called the *must synchronize* semantics of $\|$ which differs from what may be called the *may synchronize* semantics of CCS. Because of this, CCS provides another operator called *restriction* to force synchronization. The choice of must semantics then obviates the need for a restriction-like operator—at least for synchronization purposes. This is the basic communication in the pure calculus. However, PRETSEL provides a variety of communication primitives. These have their own semantic rules that differentiate, for example, blocking send operation from nonblocking send operation. These operational rules are discussed later in this section. Also, it should be noted that in PRETSEL there is not just one τ action, in fact there are a family of them—one for each possible time constraint. These τ-actions capture the time required to perform the communication.

Next consider rule (21). According to it, a temporally constrained process is capable of doing the same action as its component process as long as it is constrained meaningfully. Furthermore, in this case the temporal constraint of the resulting process is reduced by the execution time of the action involved. Rule (22) is similar,

$$\frac{P_1 \xrightarrow{e} P_1'}{P_1 \Rightarrow P_2 \xrightarrow{e} P_1' \Rightarrow P_2} \tag{14}$$

$$\frac{P_1 \xrightarrow{e} done}{P_1 \Rightarrow P_2 \xrightarrow{e} P_2} \tag{15}$$

$$\frac{P_1 : t_1 \qquad P_2 : t_2 \qquad P_1 \xrightarrow{e} P_1' \qquad (t_1 \leq t_2)}{P_1 + P_2 \xrightarrow{e} P_1'} \tag{16}$$

$$\frac{P_1 : t_1 \qquad P_2 : t_2 \qquad P_2 \xrightarrow{e} P_2' \qquad (t_2 \leq t_1)}{P_1 + P_2 \xrightarrow{e} P_2'} \tag{17}$$

$$\frac{P_1 \xrightarrow{e} P_1' \qquad P_2 \not\xrightarrow{\bar{e}}}{P_1 \| P_2 \xrightarrow{e} P_1' \| P_2} \tag{18}$$

$$\frac{P_1 \not\xrightarrow{\bar{e}} \qquad P_2 \xrightarrow{e} P_2'}{P_1 \| P_2 \xrightarrow{e} P_1 \| P_2'} \tag{19}$$

$$\frac{P_1 \xrightarrow{e} P_1' \qquad P_2 \xrightarrow{\bar{e}} P_2'}{P_1 \| P_2 \xrightarrow{\tau} P_1' \| P_2'} \tag{20}$$

$$\frac{P \xrightarrow{e} P' \qquad P : t \qquad t \leq t'}{\Omega t' : P \xrightarrow{e} \Omega(t' - \delta(e)) : P'} \tag{21}$$

$$\frac{P \xrightarrow{e} P' \qquad P : t \qquad t \leq t'}{\Pi^{t'} P \xrightarrow{e} P' \Rightarrow \Pi^{t'} P} \tag{22}$$

Figure 9: Level 1 Operational Rules

that is, a periodic process does the actions of its body process as long as the period is meaningful and it repeats forever. Rule (23) is an action axiom. According to it the computation terminates once the only action has occured. Rule (24) states that the operator & is a synchronous parallel combinator and thus both components must be willing to engage in the same action (which need not be a communication). Rules (25) and (26) are similar to rules (14) and (15). They describe the meaning of sequential composition at the task level. Rule (27) is similar to rule (22) at the process level. Rules (28)-(31) give the familiar operational meaning to the *if* and the *while* operator. Rule (32) is similar to rule (21) at the process level. Rule (33) is similar to the usual operational semantics of value-passing. Although, unlike value-passing, the value space Val_{sys} of system dependent parameters will normally be finite in practice.

The above discussion described temporal rules to capture the timing requirements of a given process or a task and also gave a transition relation that describes how a process executes and how much time it takes in its execution. The following proposition relates the temporal rules to the transition rules.

Proposition 1 *Let P be a process and let $P \xrightarrow{a_1} P_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} done$. Then $P : \sum_{i=1}^{n} \delta(a_i)$.*

Also the transition relation combines both the 'functional' behavior and the 'temporal' behavior. For non-real-time applications one may just be interested in only the functional behavior. It is clear that there are extra overheads involved in the combined behavior as one must ascertain, for example, that the processes are time correct. So, the question is whether to 'turn-off' the temporal behavior and use the operational rules for just the functional behavior without the overhead. It turns out that the answer to this question is affirmative and is summed up in the proposition below. The answer relies on an erasure mapping that erases all the timing information and what is left remaining is only the functional part. Formal details of this erasure mapping $\mathcal{E}$ are left for future work; however, just to give an idea of $\mathcal{E}$, define a erasure on actions that strips off the timing information. This is then extended to terms and the rules.

Proposition 2 *Let P be a process and $\mathcal{E}$ be the erasure mapping described above. If P terminates so does $\mathcal{E}(P)$.*

It is worth noting that the converse of the above statement may not hold in general. This is because, in the presence of time, the operator + behaves 'more deterministically' than in the absence of it.

Operational Rules for Communication Primitives

PRETSEL provides both blocking as well as non-blocking [41, 42] communication primitives. The former being synchronous in nature is used for message passing at level 2, whereas the latter being asynchronous in nature is used at level 1. In addition, primitives for pure sunchronization at both levels are provided. Figure 11 contains the operational rules for blocking send/receive primitives. Figure 12 contains operational rules for non-blocking send/receive primitives. Figure 13 contains operational rules for pure synchronization primitives. According to rule (34), if T_1 in the composite $T_1 \,\&\, T_2$ is ready to send but its counterpart T_2 is not ready to receive, then T_1 is essentially blocked—the only possible action is that of T_2. Similarly, rule (34) captures the case where T_1 is ready to receive but T_2 is not ready to send and hence T_1 must block. Rules (36) and (37) are symmetric to rules (34) and (35). Finally, when either T_1 is ready to send and T_2 is ready to receive or vice versa, the communication takes place. This is captured in rules (38) and (39). In the non-blocking case, if P_1 in the composite $P_1 \| P_2$ is ready to send a message, it sends it whether P_2 is ready to receive or not. Note that, from implementation point of view, this message is buffered if there is no matching receive at the time of sending. Rule (41) states the same for the case where P_2 is ready to send. According to rule (42), if P_1 is ready to receive, it does so without waiting for a matching send. In the case where P_2 is ready to receive, rule (43) applies. Note that if the receive occurs before a matching send, the received value may be undefined. Finally, in the rare instance where both parties are ready at the same time, a genuine communication takes place. this is captured by rules (44) and (45). Rule (46) for pure synchronization states that the component

task (process) that is ready to synchronize must wait till others are ready to do so and when all of them are ready, the synchronization takes palce. This is captured in rule (47). Similarly, rules (48) and (49) capture the synchronization at level 1.

$$\frac{}{a \xrightarrow{a} done} \tag{23}$$

$$\frac{T_1 \xrightarrow{a} T_1' \qquad T_2 \xrightarrow{a} T_2'}{T_1 \,\&\, T_2 \xrightarrow{a} T_1' \,\&\, T_2'} \tag{24}$$

$$\frac{T_1 \xrightarrow{a} T_1'}{T_1 ; T_2 \xrightarrow{a} T_1' ; T_2} \tag{25}$$

$$\frac{T_1 \xrightarrow{a} done}{T_1 ; T_2 \xrightarrow{a} T_2} \tag{26}$$

$$\frac{T \xrightarrow{a} T' \qquad T : t \qquad t \leq t'}{every\ t'\ T \xrightarrow{a} T' ; every\ t'\ T'} \tag{27}$$

$$\frac{b \equiv false}{while\ b\ T \longrightarrow nil} \tag{28}$$

$$\frac{b \equiv true \qquad T \xrightarrow{a} T'}{while\ b\ T \xrightarrow{a} T' ; while\ b\ T} \tag{29}$$

$$\frac{b \equiv true \qquad T_1 \xrightarrow{a} T_1'}{if\ b\ T_1\ T_2 \xrightarrow{a} T_1'} \tag{30}$$

$$\frac{b \equiv false \qquad T_2 \xrightarrow{a} T_2'}{if\ b\ T_1\ T_2 \xrightarrow{a} T_2'} \tag{31}$$

$$\frac{T \xrightarrow{a} T' \qquad T : t \qquad t \leq t'}{within\ t'\ T \xrightarrow{a} within\ (t' - \delta(a))\ T'} \tag{32}$$

$$\forall \vec{v} \in Val_{sys}\ \frac{T[\vec{v}/\vec{x}] \xrightarrow{a} T'}{[\vec{x}]T \xrightarrow{a} T'} \tag{33}$$

Figure 10: Level 2 Operational Rules

$$\frac{T_1 \xrightarrow{send(v)} T_1' \qquad T_2 \xrightarrow{a} T_2' \qquad a \neq receive(x)}{T_1 \,\&\, T_2 \xrightarrow{a} T_1 \,\&\, T_2'} \tag{34}$$

$$\frac{T_1 \xrightarrow{receive(x)} T_1' \qquad T_2 \xrightarrow{a} T_2' \qquad a \neq send(v)}{T_1 \,\&\, T_2 \xrightarrow{a} T_1 \,\&\, T_2'} \tag{35}$$

$$\frac{T_1 \xrightarrow{a} T_1' \qquad T_2 \xrightarrow{send(v)} T_2' \qquad a \neq receive(x)}{T_1 \,\&\, T_2 \xrightarrow{a} T_1' \,\&\, T_2} \tag{36}$$

$$\frac{T_1 \xrightarrow{a} T_1' \qquad T_2 \xrightarrow{receive(x)} T_2' \qquad a \neq send(v)}{T_1 \,\&\, T_2 \xrightarrow{a} T_1' \,\&\, T_2} \tag{37}$$

$$\frac{T_1 \xrightarrow{send(v)} T_1' \qquad T_2 \xrightarrow{receive(x)} T_2'}{T_1 \,\&\, T_2 \xrightarrow{\tau} T_1' \,\&\, T_2'[v/x]} \tag{38}$$

$$\frac{T_1 \xrightarrow{receive(x)} T_1' \qquad T_2 \xrightarrow{send(v)} T_2'}{T_1 \,\&\, T_2 \xrightarrow{\tau} T_1'[v/x] \,\&\, T_2'} \tag{39}$$

Figure 11: Operational Rules for Blocking Communication

$$\frac{P_1 \xrightarrow{par\_send(v)} P_1'}{P_1 \| P_2 \xrightarrow{par\_send(v)} P_1' \| P_2} \tag{40}$$

$$\frac{P_2 \xrightarrow{par\_send(v)} P_2'}{P_1 \| P_2 \xrightarrow{par\_send(v)} P_1 \| P_2'} \tag{41}$$

$$\frac{P_1[v/x] \xrightarrow{par\_receive(v)} P_1'}{P_1 \| P_2 \xrightarrow{par\_receive(v)} P_1' \| P_2} \tag{42}$$

$$\frac{P_2[v/x] \xrightarrow{par\_receive(v)} P_2'}{P_1 \| P_2 \xrightarrow{par\_receive(v)} P_1 \| P_2'} \tag{43}$$

$$\frac{P_1 \xrightarrow{par\_send(v)} P_1' \qquad P_2 \xrightarrow{par\_receive(x)} P_2'}{P_1 \| P_2 \xrightarrow{\tau} P_1' \| P_2'[v/x]} \tag{44}$$

$$\frac{P_1 \xrightarrow{par\_receive(x)} P_1' \qquad P_2 \xrightarrow{par\_send(v)} P_2'}{P_1 \| P_2 \xrightarrow{\tau} P_1'[v/x] \| P_2'} \tag{45}$$

Figure 12: Operational Rules for Non-Blocking Communication

$$\frac{T_i \xrightarrow{barrier} T_i' \qquad T_j \xrightarrow{a} T_j' \qquad a \neq barrier \qquad 1 \leq j \leq n \wedge j \neq i}{T_1 \,\&\, T_2 \cdots \&\, T_j \cdots \&\, T_n \xrightarrow{a} T_1 \,\&\, T_2 \cdots \&\, T_j' \cdots \&\, T_n} \tag{46}$$

$$\frac{T_i \xrightarrow{barrier} T_i' \qquad 1 \leq i \leq n}{T_1 \,\&\, T_2 \cdots \&\, T_j \cdots \&\, T_n \xrightarrow{\tau} T_1 \,\&\, T_2 \cdots \&\, T_j' \cdots \&\, T_n} \tag{47}$$

$$\frac{P_i \xrightarrow{global\_sync} P_i' \qquad P_j \xrightarrow{e} P_j' \qquad a \neq global\_sync \qquad 1 \leq j \leq n \wedge j \neq i}{P_1 \| P_2 \cdots \| P_j \cdots \| P_n \xrightarrow{a} P_1 \| P_2 \cdots \| P_j' \cdots \| P_n} \tag{48}$$

$$\frac{P_i \xrightarrow{global\_sync} P_i' \qquad 1 \leq j \leq n}{P_1 \| P_2 \cdots \| P_j \cdots \| P_n \xrightarrow{\tau} P_1 \| P_2 \cdots \| P_j' \cdots \| P_n} \tag{49}$$

Figure 13: Operational Rules for Pure Synchronization

7 PRETSEL and UNITY

In this section the PRETSEL approach is compared with the UNITY approach. UNITY is an abstract specification language for parallel programs together with logic to reason about them [9]. It is preferred to call this logic the semantics of UNITY notation. Since PRETSEL is designed for real-time applications, in addition to being a language for parallel computation, it is only appropriate to compare the functional fragment of PRETSEL with UNITY. It is possible to define *erasures* of PRETSEL specifications that would suitably eliminate all temporal and performance constructs and then define mapping between these erasures and UNITY specifications to relate them in a formal way. However, the discussion will be kept informal here and leave all formal details of this relationship as part of the future work. An informal comparision of the PRETSEL approach with the UNITY approach is given on the basis of three criteria: the basic design philosophy, the syntax, and the semantics.

The basic philosophy of the UNITY approach is to decouple a program from its implementation. Thus the emphasis is on separating concern between *what* on the one hand, and *how*, *when*, and *where* on the other. This separation results in a very simple and powerful programming notation. The basic philosophy of the PRETSEL approach too is the separation of *what* from *how*. However, the domain of the application is real-time computation and, therefore, abstraction of *when* and *where* in PRETSEL is allowed. This is because in real-time computation a distinction must be made between two programs that are functionally equivalent, but consume different resources and exhibit differing performances. Thus, for example, in the real-time domain a distinction is made between bubble-sort and quick-sort, although both are sorting procedures. Hence a real-time specification language must be based on a more intensional view of computation. As a result, performance, timing, resources, etc., which are of no importance (at the specification level) in the UNITY approach become central and essential in the PRETSEL approach. This is achieved via quantification and abstraction of various real-time features in the language. Thus, although PRETSEL has explicit timing and performance constructs, the approach has not deviated

from the basic philosophy of separation of concern of UNITY. All the real-time related constructs in PRETSEL are presented at a suitable level of abstraction that decouples a PRETSEL specification from its implementation. As with the UNITY approach, the PRETSEL approach too neither assumes nor adheres to a particular architecture.

A comparison of the syntax of UNITY programming notation with that of erasures of PRETSEL notation is presented. Although UNITY makes no explicit mention of a two level approach, the UNITY approach can be viewed as being two level—the program level and the program structuring level. These would correspond to level 2 and level 1 of PRETSEL, respectively. The basic unit of computation in UNITY is an assignment statement. The basic unit of computation in PRETSEL is an *action.* This is because PRETSEL is event-based and UNITY is state-based. However, events and states are related by a cause and effect relationship. The act of assignment in UNITY would correspond to an action in PRETSEL. PRETSEL allows for different actions since different actions may take different time or consume different resources. If the timing and performance considerations are ignored, then PRETSEL too will have ony one basic entity of computation. With regard to operators and control structures, UNITY has || and [] (although these are called separators, these can be thought of as operators) and no explicit control structure. It does, however, have an implicit looping (operationally speaking) since all assignments are assumed to execute infinitely often. The || construct of UNITY is both synchronous and asynchronous. However, in PRETSEL this operator is split into two—a synchronous operator (& at level 2) and an asynchronous operator (|| at level 1). It has been shown in [38] how to map the || of UNITY to **par** of UC [3]. The semantics of the latter are close to that of the operator & of PRETSEL. Thus, the || operator of UNITY together with a suitable mapping would correspond to the operator & of PRETSEL. The [] construct of UNITY represents a non-deterministic choice (under certain fairness assumptions). In PRETSEL, choices have been separated into two—a deterministic choice (*if* at level 2) and a non-deterministic choice (+ at level 1). It should be noted that the semantics of the latter are slightly more complex in the presence of time

and performance considerations, for it may behave as a deterministic choice operator under certain conditions. At level 2 an explicit looping construct is allowed, whereas UNITY implicitly does so. A sequencing operator $\Rightarrow$ at level 1 also has been provided. UNITY has an operator $[]$ (but the overloading of this symbol at the two levels is a bit confusing) to compose two programs. It has been shown in [38] how to define a sequencing operation on UNITY programs based on $[]$ under certain conditions. This would correspond to the sequencing operator $\Rightarrow$ of PRETSEL.

With regard to semantics, the UNITY approach is based on Hoare-style axiomatic semantics to reason about programs whereas for PRETSEL an operational semantics have been developed. This is because of the need to distinguish between computations that are deemed equivalent from a pure parallel processing point of view. Operational semantics being quite intensional allows for such distinctions to be made. In fact, it is a more general approach whereby distinctions can be made from the finest to the coarsest by defining suitable notions of equivalences.

8 Examples

In this section the applicability of PRETSEL is illustrated through three examples. The first is a simple example of vector (matrix) operations. The second is an example where the system must respond to dynamic changes in the input environment. The third example is Martin Marietta designed AN/SQS-53C Sonar System.

8.1 Vector Operations

As an example consider the following vector computations which must be completed to meet some maximum time deadline d. Let X, Y and Z be input vectors of length n, and the system must compute the two output vectors $(X+Y+Z)$ and $(X+Y)*Z$. Further, assume that the process $Q = X+Y$ is computed first and then the result vector is passed on to two concurrent processes $S = Q+Z$ and $T = Q*Z$ which compute the two desired equations. Upon completion of Q, a *parallel-send* is required to send data to the successor subtasks S and T. Let p be the total number of processors in the system. Suppose that two different algorithms T_1 and T_2 are available for computing $Q*Z$, with different execution times, and suppose T had to complete within time d_1.

Figure 14 shows the Level 2 process graph and the sample code of the process Q (as a data parallel shared memory algorithm). Figure 15 shows the formal specification for the example.

The time to complete task R, which is simply the parallel composition of processes S and T, is the maximum time to complete S and T. If Q takes more than d time then the process fails to meet the time constraints. If Q takes $d_2 < d$ time, then R must complete within $d_1 = d - d_2$ time. The task Q (and also S and T) are the data parallel algorithms, and π is the mapping function and kn/p is the execution time given as a function of n and p (thus modelling the scalability of Q). The process $\&(p)Q'$ specifies that process Q' must be executed in synchronous parallel mode on p processors; degree of parallelism in the parallel task Q is defined by the variable p (the number of processors). The process Q'' performs the arithmetic operations of

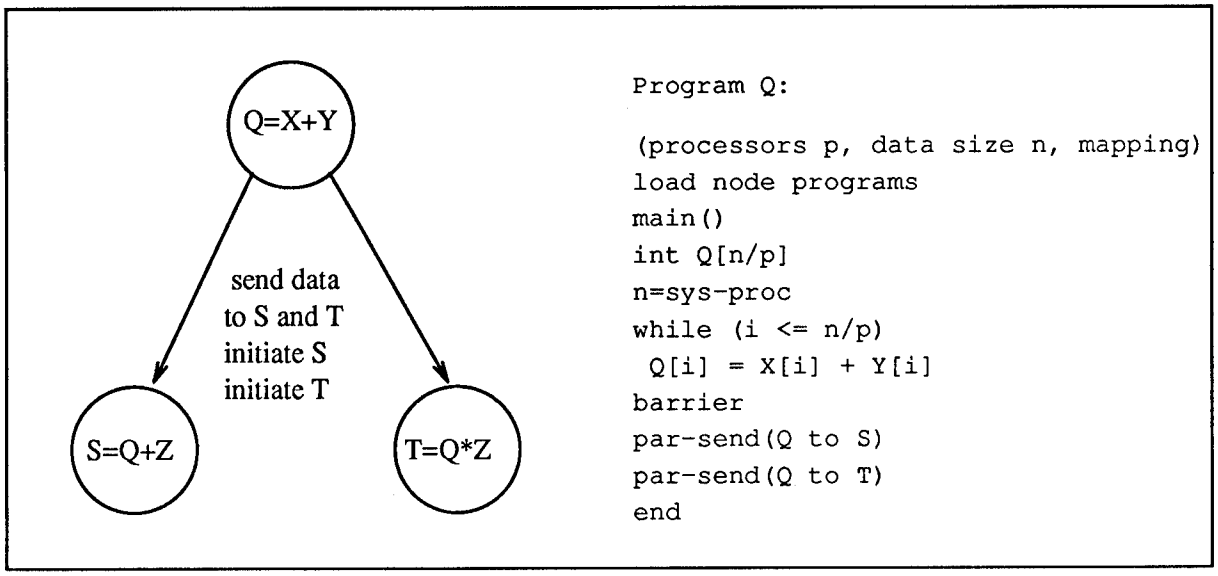


Figure 14: Example: Description of processes at each Level

$$
\begin{aligned}
&\textit{Level 1 Specification}: \\
&\qquad P \;::=\; \Omega d : Q \Rightarrow R \\
&\qquad R \;::=\; S \| T \\
&\qquad T \;::=\; \Omega d_1 : T_1 + T_2 \\
&\\
&\textit{Level2}: \\
&\qquad Q \;::=\; [n, p, \pi, kn/p] \& (p) Q' \\
&\qquad Q' \;::=\; \textit{while}\,(i < n/p) Q'' \,;\langle \textit{barrier} \rangle \,;\langle \textit{par\_send} \rangle \\
&\qquad Q'' \;::=\; \langle \textit{comp\_add} \rangle
\end{aligned}
$$

Figure 15: The Formal Specification using PRETSEL

adding the vector elements, the *barrier* is the internal communication that signals the end of the while loop, and the *par_send* is the external synchronization required to initiate process R and send data to the process S and T. The task T is specified using the timed choice operator which specifies that two different choices of algorithms are available, and that the choice is made dependent on the time taken by each algorithm based on the values of data size n and processors p available for the algorithm. The algorithm that meets the deadline, if at all, will be selected by the system as per the meaning of this operator. This illustrates how the effect of scalability of parallel algorithms is accounted for in the PRETSEL model.

8.2 Bursty IO: Dynamic Changes in the System

Next, it is illustrated how PRETSEL models a situation where dynamic changes in the input environment invoke new resource allocation processes.

Consider an example scenario, shown in Figure 16, where a sensor task S must continually receive data and then send this data to a processing task P for processing of the data. These are examples of periodic tasks since the process is repeated for each input data set that is received by the sensors. Applications where such tasks can occur include avionics where the sensor task represents data collected from radars; the change in rate of data could correspond to the use of more radars or increase in resolution. Suppose the timing constraint demands that each of these tasks must complete in t seconds to meet the real-time constraints. If a 9-processor parallel architecture is available, the processing task may be executing as a parallel algorithm using four processors in order to meet the timing requirement. At some instant in time suppose the Sensor receives a large amount of data, *i.e.,* there is a *bursty I/O*, which subsequently increases the time taken by the processing task since more data will have to be processed. Thus, the presence of bursty I/O must be specified, and the actions to be taken in its presence. To meet the timing deadline of t, the system may allocate more processors to task P to decrease its execution time. Specifically, the system must perform a re-mapping and must select a "new" algorithm (from the choices available in the table provided to the system) and this may require allocating

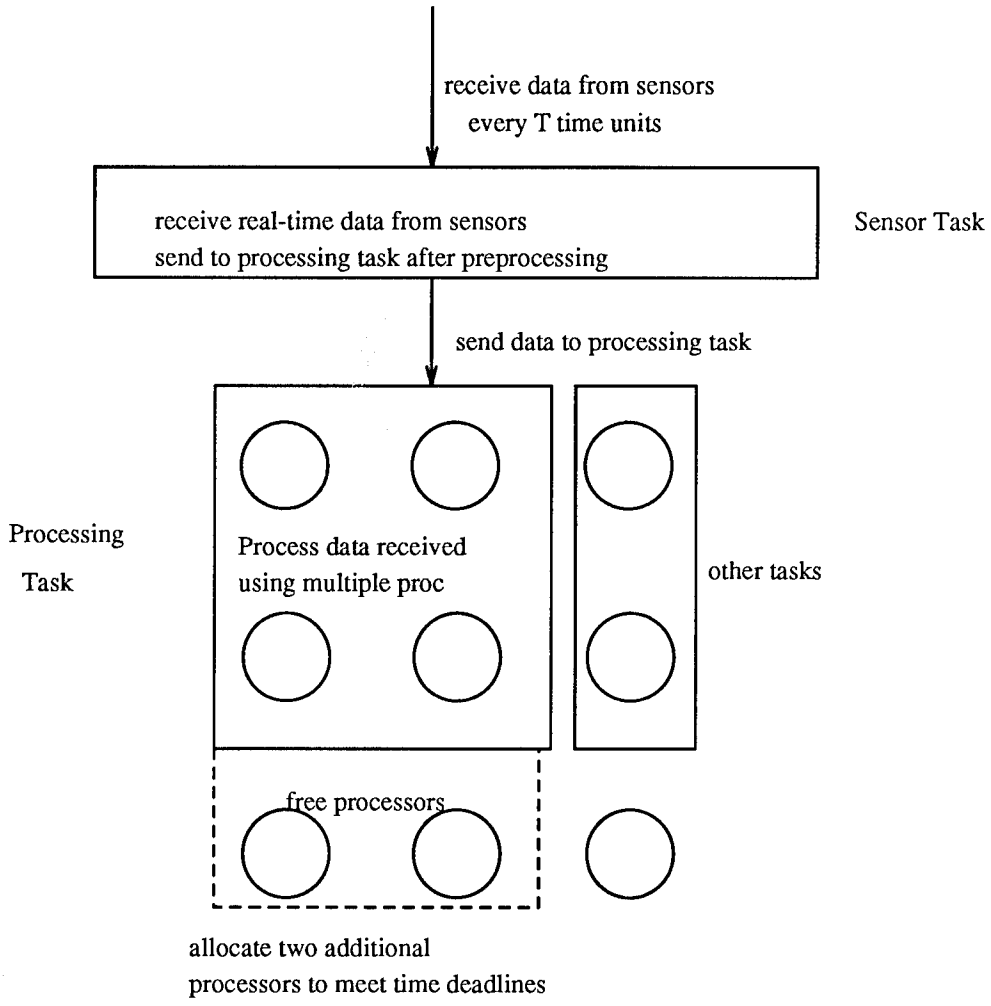


Figure 16: Example: Bursty IO handled by Architecture

two more processors to task P (by taking processors away from non-critical tasks - i.e., tasks that do not have timing deadlines). The time taken to find this new re-mapping constitutes an overhead and therefore the re-mapping process itself must be highly efficient and must meet the time constraints. Figure 17 shows the system components, behavior and the details of each of the tasks. A formal specification must capture the behavior of this process.

For this example, assume the following. There are three different algorithms that may be used to process the data; *i.e.* there is a choice of three algorithms for processing task P. The sensor task S computes the rate of arrival of data, and there is a tolerance specified for the change in data rate between two successive periods. The re-mapping task R selects and loads the code for the algorithm that must be used by P to satisfy the performance requirements.

Now consider the modelling of the above example using the PRETSEL model. The formal specification of the system is shown in Figure 18. The choice operator $P+P'$ signifies that process P or P' will be initiated; in the system implementation this will be based on the change in the I/O rate and will be initiated by the Sensor task S. When the system is first started P' is invoked to make the selection and load the code. The choice operator for the processing task P signifies that the choice of algorithms (code) for the task P will be based on which algorithm meets the time requirements. Once again, this selection is made by the re-mapping process and the selection does not change *until* the re-mapping process is invoked again. When the re-mapping task R is invoked it will determine which algorithm must be selected, and the time information is conveyed through the timed sequential operator. The maximum time deadline "$\Omega\, t$" specifies that the process P must take no more than time t to meet its periodic time constraint.

8.3 Specification of a Sonar System

In this section the effectiveness of PRETSEL in modelling a real application is demonstrated, by showing how PRETSEL can be used to specify the Martin Marietta designed AN/SQS-53C Sonar System with an emphasis on the Active Receive Beam-

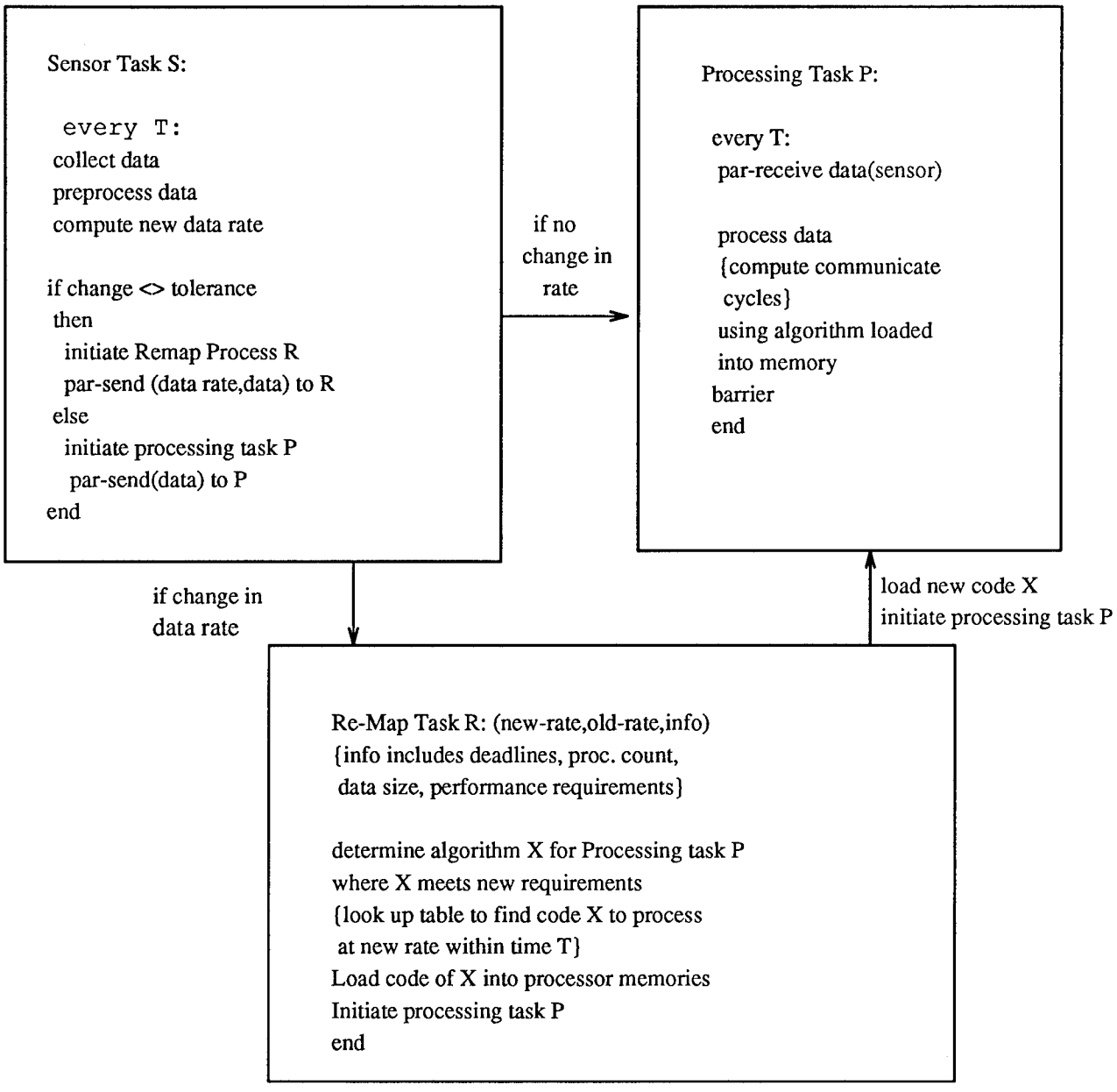


Figure 17: System Components for Handling Bursty IO

$$
\begin{aligned}
System &::= S \Rightarrow Q \\
Q &::= P + P' \\
P' &::= \Omega t : R \Rightarrow P \\
P &::= P_1 + P_2 + P_3
\end{aligned}
$$

Figure 18: Formal Specification (for Level 1) of System

former component of the system. Towards this end, the following is addressed:

- decompose the beamformer into major functional units
- consider the implementation of the Sonar on a parallel computer
- address the parallelism inherent in the different functional units
- model functionality using PRETSEL
- demonstrate the effectiveness of PRETSEL

The AN/SQS-53C sonar system was designed, developed and manufactured by Martin Marietta. It is an active and passive, hull array sonar system designed specifically to meet the requirements of modern naval vessels. A passive sonar listens for radiated noise from the environment and uses processing techniques to determine the bearing and characteristics of that radiated noise. An active sonar transmits sound at a certain frequency into the environment and processes the noise for echos from the transmitted energy.

The sonar provides three active modes of operation and one passive mode. The Surface duct (SD) mode provides a full 360 degrees of active surveillance capability. It may be operated concurrently with either the Variable depression (VD) mode and/or the track (TK) mode in addition to the passive (PA) mode. SD is used to track close-in targets and to maintain a highly competent, panaromic anti-submarine

warfare active surveillance. The Variable depression mode provides a detection capability over and up to 240 degree search sector. In VD mode both transmit and receive beams can be steered vertically as well as horizontally. The VD mode provides waveforms, processing techniques, and displays formats specifically designed to detect, track and classify submarines. The VD mode may be operated concurrently with either the SD or TK modes. The Track mode provides a narrow bearing and range window surrounding a suspected contact for highly accurate range, bearing and Doppler estimation. The track mode is initialized via operator selection from a search mode (VD or SD) once a track has been established on a given contact. Track beams are steered directly at the contact and lock onto the target as it maneuvers. The TK mode may be operated concurrently with either the SD and/or the TK mode.

The AN/SQS-53C system can be depicted simplistically by the major functions that it performs. The acoustic data arrives at the transducers, is signal conditioned and then forwarded to the active and passive receivers. The active receive processing can itself be decomposed into: Signal Conditioning, Beamforming, Signal Processing, Data Processing, and Display and Operator machine interface. The focus of the example is the Beamforming component within the active receive processing which is described in more detail in the following paragraphs. The goal is to provide a PRETSEL specification of the functionality, and the timing, of the Active Receive Beamformer component of the sonar system.

Beamforming is a technique for performing spatial filtering, *i.e.,* signals and noise arriving at the array from angles other than the array look direction are attenuated relative to coherent signals and interferences arriving at the array from the look direction. The beamformer receives the digital element data from the signal conditioner. The control data from the Controller defines the modes (SD,VD, TK), the beam bearings, sampling rates, and frequency bands to be processed. The beamforming is performed in two stages: first, vertical beamforming is performed, followed by horizontal beamforming. The beams are stabilized for own ship heading, roll and pitch. Sub-band filtering is performed following the beamforming. These sub-band filters remove the effects of the own ship doppler (called own ship doppler nullification)

and generate independent mode/waveform processing bands. Figure 19 shows the outline of the processing required within the beamformer. The resulting data (of the beamformer) is output to the Active Analyzer program.

The precise processing required for active sonar data in the Beamformer has many permutations depending on the digital input data and the control data. Factors that influence the choice of operating mode, coverage, and waveform include the ships mission, the threat, and the environmental conditions. All of these factors can vary at any time causing an impact on the amount of processing performed by the beamformer. Since the sonar is a real-time system, the system must adapt to meet the loading on the beamformer caused by the operators choice of operating mode, coverage, and waveform. This also suggests that the timing requirements need to be specified in terms of the maximum delay allowed. The time and ownship data is supplied once every 100 milliseconds; in other words the beamformer must finish processing one "data set" within 100 milliseconds. The data set itself will vary depending on the modes selected, and will consist of data collected over 512 time samples (within the 100 milliseconds). The output data to be computed varies with the task and the input data; for example the VBF forms 72 stave beams for each of the modes (SD, VD, or TK), thus generating up to 216 total beams. The Configuration Evaluation (CE) component in Figure 19 defines the step at which the parameters, such as modes and sampling rates, are set. This configuration evaluation process occurs at every "ping" of the sonar, where the "ping" cycle itself may be set by the operator. Between each ping the sonar periodically receives data that must be processed by the beamformer. Since the processing requirements may be changed by the CE task, at every "ping", a *remapping* of the computing resources must be performed to assure that the beamformer meets the real-time requirements. Based on the parameters provided by the CE the remapping task must determine the number of processors to be used for each of the beamformer components.

The outline of the Vertical Beamformer process is shown in Figure 20. Based on the coefficients supplied by the CAL process, the eight elements in a stave are summed together to form three sets of beams. It provides the necessary data buffering

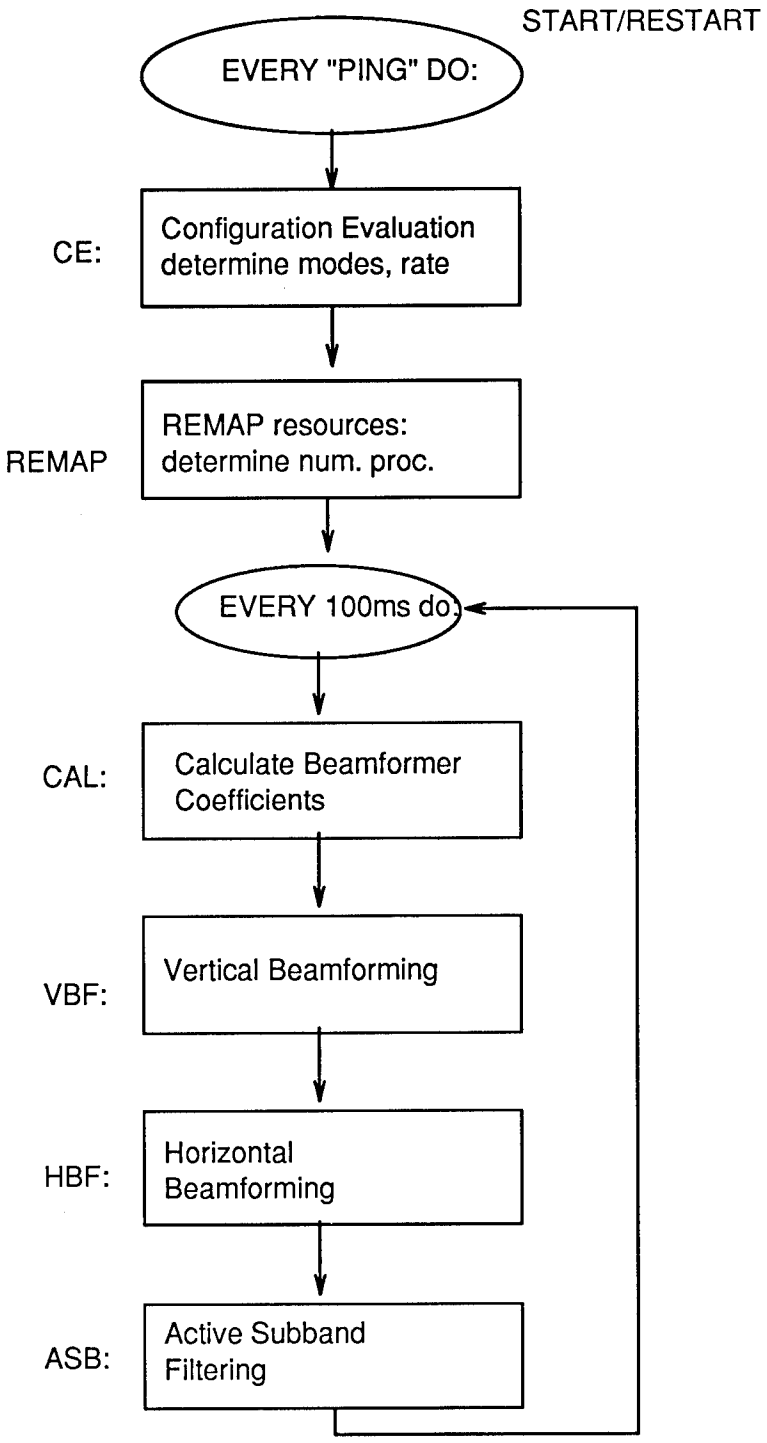


Figure 19: Flowchart of Active Receive Beamformer Process

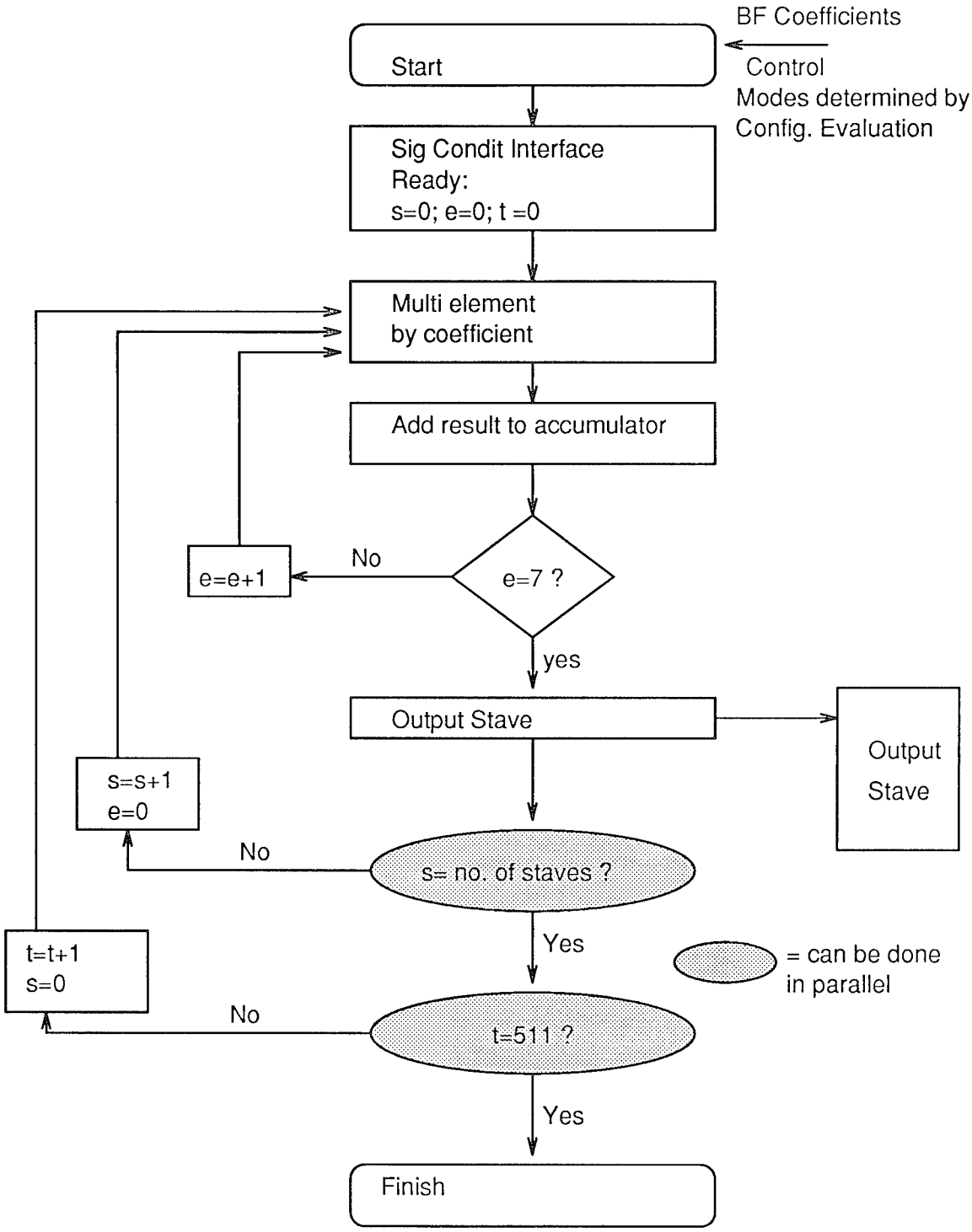


Figure 20: Flowchart of Vertical Beamformer Process

to accomodate the time delays necessary for vertical beamforming. The Horizonatal beamformer (HBF) receives its input data from the VBF. Due to the similarity between the processes, and for brevity, the details of the HBF and ASB tasks are omitted. The VBF process exhibits a large degree of data parallelism. Specifically, each of the 72 staves can be computed in parallel and also each of the 512 different samples can be computed in parallel. The dependencies exist only in the sense that two different staves will need to *read* the same input data; in other words there is no read-after-write dependencies and the entire loop in Figure 20 may be parallelized.

Using the functional decompositions shown in Figures 19 and 20, a formal specification of the Active receive beamformer process can be provided in PRETSEL. Figure 21 shows the complete Level 1 specification of the Beamformer (BEAMFORM) process depcited in Figure 19. The figure shows the complete level 2 specification of the Vertical Beamformer (VBF) process only. The other level 2 tasks can be specified in a similar manner. The following observations must be noted:

- modes to be used D_{sd}+D_{vd}+D_{tk} are determined by the Configuration evaluation (CE) task, which is invoked at every "ping" and determines the choices based on the type of objects being tracked by the sonar. The operator may also choose to operate some of the modes concurrently, and select the input sampling rate.

- the number of processors, and the specific algorithm to be used, is determined by the Remap task based on the sampling rate and the modes selected by the Configuration Evaluation task.

- D_{sd} denotes processing of Surface Duct (SD) mode using data N_{sd} for SD mode. Similarly, D_{vd} and D_{tk} are defined for the data used in the Vertical depression and the Track modes.

- The input data to the Vertical Beamformer(VBF) is generated by the previous task which calculates the coefficients (CAL).

- The data computed by the VBF task is sent to the Horizontal Beamformer task (HBF) using the *par_send* command.

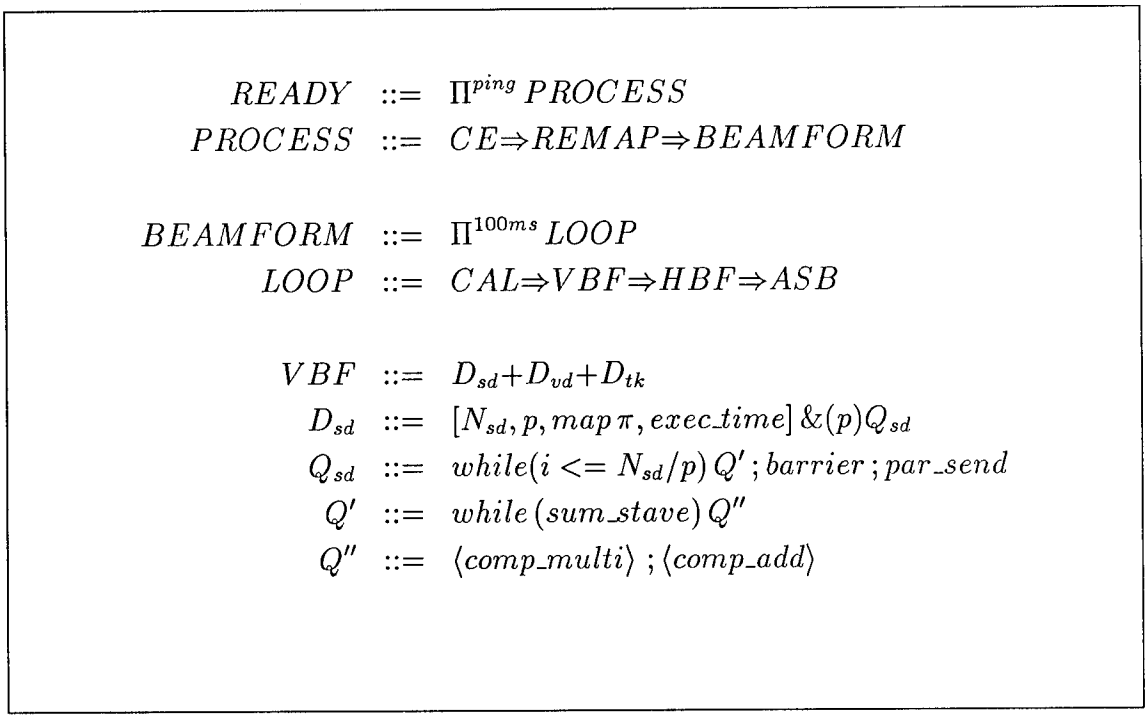


Figure 21: Formal Specification of Active Receive Beamformer

$$
\begin{aligned}
READY &::= \Pi^{ping}\, PROCESS \\
PROCESS &::= CE \Rightarrow REMAP \Rightarrow BEAMFORM \\
BEAMFORM &::= \Omega X : LOOP \\
\\
LOOP &::= CAL \Rightarrow VBF \Rightarrow HBF \Rightarrow ASB \\
\\
VBF &::= D_{sd} \| D_{sd} \| D_{tk} \\
D_{sd} &::= [N_{sd}, p, map\,\pi, exec\_time]\, \&(p) Q_{sd} \\
Q_{sd} &::= every\, 100ms\, P \\
P &::= while\, (i <= N_{sd}/p)\, Q' \; ; \; barrier \; ; \; par\_send \\
Q' &::= while\, (sum\_stave)\, Q'' \\
Q'' &::= \langle comp\_multi \rangle \; ; \langle comp\_add \rangle
\end{aligned}
$$

Figure 22: Specification of Pipelined Execution

The specification defined in Figure 21 did not provide a pipelined implementation. A pipelined implementation of the beamformer process can be specified using PRET-SEL. By specifying the Level 2 tasks as periodic processes, with a period of 100ms, the level 1 tasks we can specify a pipelined execution. In the beamformer, there is also a constraint on the maximum time taken to process each data set by all the tasks. In other words, a response time (latency) constraint X ms, time to process each data set, is imposed. The pipelined specification is shown below in Figure 22. Note that each level 2 task is specified as a periodic task. One of the system issues introduced by this scenario is that of resource allocation schemes, to assign processors/memory, which can balance throughput and the response time constraints.

9 Summary and Future Work

This report discussed the problem of formal specification of real-time systems implemented on a parallel machine.

For many real-time applications parallel computers offer a natural computing platform and offer an attractive method to place higher processing requirements, due to more sensors or additional information, on real-time systems. However, the lack of software support both in the design as well as in the implementation phases has resulted in a slower acceptance of parallel computing than originally expected. Furthermore, very little attention, if at all, has been paid to real-time embedded system requirements. The general goal of this research project was to investigate important issues related to the design, development and validation of real-time system software for parallel computers. In particular, the objectives were to consider formal models for specification of real-time systems implemented on parallel computer systems. A formal specification language will allow the system designer to specify the structure of the real-time system and make timing assertions about the system, while leaving the complex problems of resource allocation and verification to automation. The provision of such a formal specification model with a well-defined syntax and semantics will allow the development of automated verification tools.

A number of researchers have provided formal specification models, and verification methodologies, for real-time systems on conventional machines but there has been minimal work on real-time systems implemented on parallel machines. The problem of specification, design, and analysis of real-time systems and software (including issues such as specification, language design, compiler support, and operating systems) is made more difficult by the concurrency in real-time applications and further complicated by the presence of time. This report provided a review of current formal methods for specification of real-time systems and evaluated their expressive power, or lack thereof, in specifying parallel real-time systems.

This project proposed a specification language PRETSEL (Parallel REal-Time SpEcification Language) for parallel real-time systems. The PRETSEL model incor-

porates some features of CSP, CCS, and process algebras while providing additional constructs to express the issues introduced by parallelism such as scalability and degree of parallelism.

The PRETSEL specification language is based on a traditional two-level computing model for parallel computing whereby a parallel computation is viewed as a collection of interacting (data) parallel algorithms. This view is naturally reflected in PRETSEL syntax where at the lower level various constructs are provided for the specification of a data-parallel real-time algorithm (data-parallelism). At the upper level another set of constructs is provided to combine such tasks in a variety of ways (task-parallelism). Furthermore, the PRETSEL language allows for the specification of performance requirements. This is achieved by allowing parameterization of tasks by system related performance specifications. This includes, for example, number of processors, execution time information, mapping specification, etc. PRETSEL clearly maintains a separation between functional requirements, temporal requirements, and performance requirements. It provides temporal scope constructs to be able to specify periodic tasks and also tasks with hard and soft deadlines. It also supports a variety of communication mechanisms. A formal operational semantics of PRETSEL has been defined and some relevant results have been established. The relationship of PRETSEL's syntax and semantics to other existing models has also been discussed. The PRETSEL model has been used to specify the behaviour of a Sonar Beamformer, which is part of the Martin-Marrietta sonar system, using off the shelf components. This example clearly demonstrated the effectiveness of PRETSEL in the specification of an actual real-time system.

9.1 Future Work

A number of issues can be explored to bring to fruition our endeavor of developing a robust parallel real-time system.

- Extend the PRETSEL specification language and model to incorporate heterogeneuty, fault-tolerance and exception handling.

- The table driven approach should be further developed to provide a complete set of system requirements for the PRETSEL model.
- Proof system for Verification: To develop the verification tool, establish a transition model and provide a proof system. Efforts shall be directed towards establishing the soundness and completeness of the proof system.
- Investigate automatic synthesis and rapid prototyping of real-time programs in parallel and distributed environments.
- Design of a PRETSEL based Validation, Verification and Synthesis Toolkit.
- System software issues such as provision of a run-time support for communication, scheduling and resource allocation algorithms, and operating system support.

References

[1] R. Alur and T. A. Henzinger, Real-time logics: complexity and expressiveness. *Proc. Fifth Annual Symposium on Logic in Computer Science*, 390–401, IEEE Computer Society Press, 1990.

[2] J.C.M. Baeten and J.A. Bergstra, Real Time Process Algebras, *Technical Report CS-R9053, Centre for Mathematics and Computer Science,* Amsterdam, The Netherlands, 1990.

[3] R. Bagrodia, K.M. Chandy, and E. Kwan, UC: A Language for the connection machine, *Proc. Supercomputing 1990*, 525–534, 1990.

[4] A. Bernstein and P. K. Harter, Proving real-time properties of programs with temporal logic, *Proc. ACM SIGOPS 8th Annual ACM Symposium Operating System Principles,* Dec. 1981, 1-11.

[5] G. Berry and L. Cosserat, The Esterel synchronous programming language and its mathematical semantics, *Lecture notes in computer science*, Vol. 197, 389–448, Springer-Verlag, 1985.

[6] J. Blazewicz, Deadline scheduling of tasks with ready times and resource constraints, *Information Proc. Letters,* Vol. 8, No. 2, Feb. 1979.

[7] T. Bolognesi and F. Lucidi, LOTOS-like process algebra with urgent or timed interactions. K. Parker and G. Rose, editors, *Proceedings of the Fourth International Conference on Formal Description Techniques (FORTE'91),* North-Holland, November, 1991.

[8] P. Brinch Hansen, *Parallel Programming Paradigms*, Prentice-Hall, 1995.

[9] K. M. Chandy and J. Mishra, *Parallel Program Design: A Foundation,* Addison-Wesley, 1988.

[10] J. E. Coolahan, Timing requirements for time-driven systems using augmented Petri nets, *IEEE Trans. Software Eng.,* SE-9(5):603–616, (September 1983).

[11] B. Dasarathy, Timing constraints of real-time systems, *IEEE Trans. Software Eng.,* SE-11(1):80–86, (January 1985).

[12] J. Davies and S. Schneider, An Introduction to Timed CSP, Technical Report, PRG, Oxford, 1989.

[13] J. Davies, *Specification and Proof in Real-Time CSP*, Cambridge University Press, 1993.

[14] N. Dershowitz, Orderings for term-rewriting systems, *Theoretical Computer Science,* Vol. 17, 279–301, 1979.

[15] R.B.K. Dewar, G. A. Fisher, E. Schonberg, R. Froehlich, S. Bryant, C. F. Goss, and M. G. Burke, The NYU Ada translator and interpreter, *Proc. IEEE COMPSAC'80*, October 1980.

[16] R.B.K. Dewar, E. Dubinsky, E. Schonberg, and J.T. Schwartz, Programming with Sets: An Introduction to the SETL Programming Language, Springer-Verlag, 1986.

[17] A. Diller, *Z: An Introduction to Formal Methods*, John Wiley and Sons, 1990.

[18] R. Duke and G. Smith, Temporal logic and Z specifications, *The Australian Computer Journal*, 21(2):62–66, 1989.

[19] C. J. Fidge, Specification and verification of real-time behaviour using Z and RTL, *Proc. Formal Techniques in Real-Time and Fault-Tolerant Systems*, LNCS Vol 571, 393–408, Springer-Verlag 1991.

[20] R. P. Gabriel (ed.), Draft report on requirements for a common prototyping system, *ACM SIGPLAN Notices* 24(3):93–165, 1989.

Springer-Verlag, New York, 1986.

[21] A. Galton, Editor, Temporal Logics and their Applications, New York: Academic Press, 1987.

[22] V. Gehlot, Performance specification and livelock detection/correction of a protocol using timed Petri nets, *Proc. International Conference on Communications*, 1286–1290, 1988.

[23] V. Gehlot and I. Lee, 'Formal specification and analysis of DMI—an x.25 based protocol, *Proc. IEEE INFOCOM'88*, 641–650, 1988.

[24] R. Gerber and I. Lee, A Proof System for Communicating Shared Resources, *Proc. of IEEE Real-Time Systems Symposium,* 288-299, 1990.

[25] M. Hennessy and T. Regan, A Temporal Process Algebra, *Technical report 2/90,* University of Sussex, UK, April 1990.

[26] M. Hennessy and T. Regan, A Process Algebra for Timed Systems, Technical Report 5/91, Computer Science, University of Sussex, Brighton, April 1991.

[27] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[28] J. Hooman and J. Widom, A temporal logic based compositional proof system for real-time message passing, *Dept. of Comp. Science technical report 88-919,* Cornell University, Ithaca, NY, 1988.

[29] S.F. Hummel and R. Kelly, A Rationale for Massively Parallel Programming with Sets, *Journal of Programming Languages*, 1 (3).

[30] F. Jahanian and A. K.-L. Mok, Safety analysis of timing properties in real-time systems, *IEEE Trans. Software Eng.*, SE-12(9):890–904, (September 1986).

[31] K. B. Kenny and K.-J. Lin, Building flexible real-time systems using the Flex language, *IEEE Computer,* Vol. 24, No. 5, 70–78 (May 1991).

[32] A. S. Klusener, Completeness in Real Time Process Algebras, *Technical report CS-R9106, Centre for Mathematics and Computer Science,* Amsterdam, The Netherlands, January 1991.

[33] R. Koymans et al., Compositional semantics for real-time distributed computing, *Lecture notes in computer science*, Vol. 193, 167–189, Springer-Verlag, 1985.

[34] R. Koymans, J. Bytopil, and W. P. de Roever, Real-time programming and asynchronous message passing, *Proc. 2nd Symposium Principles of Distributed Computing,* Montreal, Aug, 187–197.

[35] I. Lee and V. Gehlot, Language constructs for distributed real-time programming, *Proc. Real-Time Systems Symposium*, 57–66, San Diego, California, December 1985.

[36] I. Lee, P. Brémond-Grégoire, and R. Gerber, A process algebraic approach to the specification and analysis of resource-bound real-time systems, *Proc. of the IEEE,* pp. 158–171, vol.82, No.1, Jan.1994.

[37] J. W.S. Liu, K-J. Lin, W-K. Shih, and A. C. Yu, Algorithms for scheduling imprecise computations, *IEEE Computer,* Vol. 24, No. 5, 58–68 (May 1991).

[38] Y. Liu, A. K. Singh, and R. L. Bagrodia, A decompositional approach to design of parallel programs, *IEEE Trans. Soft. Engineering,* SE-12(20), 914–932, December 1994.

[39] Z. Manna and A. Pnueli, How to cook a temporal proof system for your pet language, *Proc. Symposium Principles of Programming Languages,* Austin, TX, Jan. 1983, 141–154.

[40] T. Martin, Real-time programming language PEARL—concepts and characteristics, *Proc. COMPSAC*, 301–306, Chicago, Illinois, 1978.

[41] Message Passing Interface Forum, Document for a standard message passing interface, Technical report, University of Tennessee, Knoxville, Tenn., 1994.

[42] Message Passing Interface Forum, MPI: a message passing interface, *Proc. Supercomputing'93*, 878–883, 1993.

[43] R. Millner, Communication and Concurrency, Prentice-Hall, 1989.

[44] R. Milner, A Calculus of communicating systems, *Lecture notes in Computer Science 92,* Springer-Verlag, 1980.

[45] R. Milner, Calculi for synchrony and asynchrony, *Theoretical Computer Science,* Vol. 25, 267–310, 1983.

[46] A. Mok, SARTOR – A design environment for real-time systems. *Proc. of the 9th IEEE COMPSAC,* 174–181, 1985.

[47] F. Moller and C. Tofts, A temporal calculus of communicating systems, *Proc. of CONCUR'90, LNCS 458*, 401–415, 1990, Springer-Verlag New York.

[48] X. Nicollin and J. Sifakis, An Overview and Synthesis of Timed Process Algebras, *Proceedings of the REX Workshop on Real-Time: Theory in Practice, LNCS 600,* 1991.

[49] X. Nicollin, J.L. Richier, J. Sifakis, and J. Voiron, ATP: an Algebra for Timed Processes, *Proceedings of the IFIP TC 2 Working Conference on Programming Concepts and Methods,* Sea of Gallilee, Israel, April 1990.

[50] X. Nicollin and J. Sifakis, The algebra of timed processes ATP: theory and applications, *Information and Computation,* Dec. 1990.

[51] X. Nicollin, J. Sifakis, and S. Yovine, From ATP to Timed Graphs and hybrid systems, in J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *LNCS 600, Proceedings of REX Workshop "Real-time: Theory in Practice",* The Netherlands, Springer-Verlag, June 1991.

[52] J. S. Ostroff, Formal Methods for the Specification and Design of Real-Time Safety Critical Systems, *Journal of Systems and Software,* 33–60, April 1992.

[53] J. S. Ostroff, *Temporal Logic of Real-time Systems*, Research Studies Press, 1990.

[54] J. S. Ostroff and W. M. Wonham, A framework for real-time discrete event control, *IEEE Transactions on Automatic Control*, Vol 35, No. 4, April 1990, 386–397.

[55] J.S. Ostroff, Temporal logic and extended state machines in discrete control, in M. J. Denhaum and A. J. Lamb, editors, *Advanced Computing Concepts and Techniques in Control Engineering, Vol. 47 of NATO ASI Series F: Computer and Systems Sciences,* New York Springer-Verlag, 1988, 213–236.

[56] J.S. Ostroff, A temporal logic approach to real-time control, *Proc. 24th IEEE Conference Decision Control*, Florida, Dec. 1985, 656–657.

[57] J. S. Ostroff and W.M. Wonham, Modelling, specifying and verifying real-time embedded computer systems, *Proc. 8th IEEE Real-time Systems Symposium,* San Jose, CA, Dec. 1987, 124–132.

[58] A. Pnueli, The temporal logic of programs, *Proc. 18th IEEE Annual Symposium Foundations of Computer Science,* Providence, RI, Nov. 1977, 46–57.

[59] A. Pnueli, Application of temporal logic to the specification and verification of reactive systems: a survey f current trends, *Lecture notes in computer science*, Vol. 224, 510–583, Springer-Verlag, 1986.

[60] K. Ramamritham, J.A. Stankovic, and P-F. Shiah, Efficient Scheduling Algorithms for Real-time Multiprocessor Systems, *IEEE Trans. Parallel and Distributed Systems,* Vol. 1(2): 184–194 (April 1990).

[61] G.M. Reed and A. W. Roscoe, A timed model for communicating sequential processes, *Theoretical Computer Science*, 58, 249–261, 1988.

[62] J. T. Schwartz et al., *Programming with Sets: An Introduction to SETL*, Springer-Verlag, 1986.

[63] R. L. Schwartz and P.M. Melliar-Smith, From state machines to temporal logic: specification methods for protocol standards, *IEEE Transactions on Communications,* Vol. COM-30, Dec. 1982.

[64] A. Shaw, Reasoning About Time in Higher-Level Language Software, Technical Report 87-08-05, Department of Computer Science, University of Washington, Seattle, August 1987.

[65] A. D. Stoyenko, A schedulability analyzer for real-time Euclid, *Proc. Real-Time Systems Symposium*, 218–227, 1987.

[66] Andre M. van Tilborg and Gary M. Koob, editors, *Foundations of Real-Time Computing: Formal Specifications and Methods,* Kluwer Academic Publishers, 1991.

[67] C. Tofts, Temporal Ordering for Concurrency, University of Edinburgh Report No. LFCS-88-49, 1988.

[68] W. Yi, Real-time behaviour of asynchronous agents, in J.C.M. Baeten and J.W. Klop, editors, LNCS 458, *Proceedings of CONCUR '90,* The Netherlands, 502–520, Springer-Verlag, August 1990.

[69] W. Yi, CCS+Time = an interleaving model for real-time systems, *Proceedings of ICALP'91,* Madrid, Spain, July 1991.

[70] W. Zhao, K. Ramamritham, and J.A. Stankovic, Scheduling tasks with resource requirements in hard real-time systems, *IEEE Trans. Software Engineering,* Vol. SE-12, May 1987.

*U.S. GOVERNMENT PRINTING OFFICE: 1996-710-126-20220

MISSION

OF

ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.