Progress Report for Oct 1, 1988 to Feb 28, 1989

K.T.Narayana
Department of Computer Science
Whitmore Laboratory
The Pennsylvania State University
University Park, Pa 16802

The Office of Naval Research under its programme of "fundamental research initiatives in real-time computing" sponsored my research into Real-Time System: Specification and Verification by a grant number N00014-89-J-1171.

During the period from Oct 1, 1988 to Feb 28, 1989 the following progress have been made with regard to the goals of the project.

1 Real-Time System Specification

Temporal Interval Logic by virtue of its abstract interval constructors offers an excellent setting for concurrent system specification. Making extensions for this logic to cater to specification of quantitative aspects of behaviors was considered useful. Extensions have been made to for specifying real-time constraints in the framework of the logic. A formal semantics has been given. A large body of realistic and nontrivial examples have been explored and specified. The examples considered suggest that real-time variants of temporal interval logic are extremely suitable for quantitative specification of behaviors. The work was published at the 9th symposium on real-time systems, held in December 1988 at Huntsville, Alabama. A revision of the paper is currently under way so as to submit it for a journal.

The work needs to be explored further in the direction of decision procedures for finite state real-time controllers. In an earlier work we proved that temporal interval logic is PSpace Complete. We have obtained decision procedures and implementations. We have synthesized using our implementation a hardware arbiter. This in itself is a substantial acheivement for the synthesis times in our Prolog based implementation were quite reasonable. Thus the logic was demonstrated to be useful. We continue the direction of our research into synthesis by conducting experiments on synthesis of several finite state systems. We hope to report our experiences in the due course of time, both in the use of the logic and also on the technique of synthesis. We seek to study decision procedures for real-time variants of the logic. The central problem that needs to be addressed is the encoding of transitions. For in a real-time system specification, the number of transitions generated in the synthesis are extremely high and storage and search will be at a premium. We are currently exploring these encoding aspects.

2 A Limited Parallelism Semantic Model for Real-time Concurrency

Concurrency has been investigated in the literature for a long time both from a linguistic point of view and from a semantic point of view. The foundational work of Dijkstra on concurrent programming was the centerpiece of all later efforts in concurrency theory. Dijkstra introduced the linguistic notion of a *semaphore* for eliminating time dependent errors in concurrent programs. Viewed in a semantic framework, a semaphore is a linguistic mechanism for implementing atomic actions. Thus the syntactic notion of atomic actions (specified by matching "<" and ">" brackets enclosing a piece of text in which shared variables are modified) is the main stay of Dijkstra's work. A discipline requiring that every shared variable in the concurrent program be manipulated inside a bracketed section facilitates the elimination of time dependent errors when an appropriate semantics is imposed on the bracketed sections. Thus ensuring that the intermediate states of an atomic action are unobservable enforces the requirement that atomic actions of processes which have common shared variables must be interleaved during an execution. Viewed in a different manner, atomicity imposes constraints on what a process scheduler can and cannot do.

As a result of the semantic prescription, we arrive at an interleaving model of concurrency. This interleaving concurrency framework imposes minimal restrictions on the scheduler, though there is an implicit imposition that the scheduler must eventually schedule an action. Further, it offers a useful executional abstraction, relieving the programmer from considering the delays associated with the scheduling of actions. However, interleaving model is a weak semantic model in that one cannot infer strong properties of the execution model even if the execution model were precise.

Since a quantitative concurrency model is a must for real-time concurrency, interleaving models are inadequate. This brings us to question not only the sufficiency of the available linguistic elements for concurrency, namely atomic actions, but also to seek quantitative models which do not *excessively* restrict implementation choices. The first casualty of such models would be that scheduling delays (namely context switching times) have to be accounted for in some manner. This is in contrast to the interleaving models in which we can safely ignore such delays. Thus certain nice aspects would necessarily be lost.

To alleviate the problem of taking into account context switching delays, Koymans *et. al.* advocate a model for real-time computing which makes schedulers irrelevant. It is called the *maximum parallelism* model, an adaptation from Salwicki and Müldner. The model requires that each process in the program be executed on a separate processor. A global clock (discrete) relates events quantitatively. Thus maximum parallelism model equates the execution model with the semantic model. It is the strongest model that one can offer for real-time computing. However, in a realistic situation, it is generally the case that there will be fewer processors than processes in the program. Thus maximum parallelism model is *unrealistic*. A semantic model must therefore cater to limited parallelism rather than to maximum parallelism.

When limited parallelism is the basis, a wide variety of issues arise with regard to formal modeling of real-time concurrency. *Scheduling theorists* view real-time concurrency as a scheduling problem. Their work is rooted on the notion that a real-time program can be split into tasks. A task and process are distinct concepts. For each of the tasks in the program, certain *deadlines* are specified. They assign the scheduler the responsibility of meeting those deadlines.

We question this approach. *Is it the responsibility of a scheduler to guarantee the quantitative evolution of program state or that of the programmer to code his application in such a way that, from the semantics of the language and the text of the program, he can infer that the program meets the required quantitative aspects in regard to the evolution of the program state?*

Scheduling theorists in effect argue that management of processors under certain timing constraints is an operating system function. No known formal model of *real-time* concurrency has delegated the processor management function as an operating system function. For concurrency *minus real-time*, we should note that the interleaving model, by virtue of executional abstraction, *delegated* processor management as an operating system function. The **THE** multiprogramming system has been the corner stone for such delegation of processor management as an operating system function.

If the programmer specifies deadlines, then several issues arise. Tasks can themselves be concurrent programs; further several individual tasks can be concurrent (note that we refer here to cooperation between concurrent tasks). Thus, it is not clear under what formal model the programmer characterizes the deadlines for tasks, even if they are worst-case deadlines. It is this aspect which casts doubts on the general applicability of processor management *exclusively* by an operating system and without any direction from the program. In addition, becuase of inherent problems associated with scheduling algorithms (optimal scheduling is *NP-hard*), there arise notions of *missed deadlines.* The notion of *missed deadlines* causes correctness problems. Even for the restricted classes of programs considered by them, the implementation makes a visible impact on the semantic characterization of quantitative concurrency at a higher level. This will be the case if we regard the processor management model as the basic underlying model upon which the concurrency model is built. Such close integration of the underlying system with the program makes the formal model extremely complex. It is difficult to conceive that in such a model proofs of programs can be established with *sufficient confidence.*

Thus we are faced with a situation in which maximum parallelism model is unrealistic, and processor management for limited parallelism exclusively by an operating system is *at best unwieldy*. As a result we need a semantic model for limited parallelism which is stronger than the interleaving model, weaker than the maximum parallelism model, and at the same time does not unnecessarily restrict implementation choices. This semantic model must facilitate reasoning about the quantitative evolution of the program state, for that is the fundamental requirement of any real-time concurrency model.

2.1 The Limited Parallelism Model

Recent work by Shade and Narayana provides a semantic model for real-time concurrency. It considers a small low level and representative concurrent programming language $\mathcal{L}$. The language uses shared variables for process cooperation, a bus arbitration model of shared memory access, a command for process synchronization, atomic actions for constructing critical sections, and a delay command for real-time control. In the limited parallelism model, the formalization caters to

- maximizing truly concurrent activity at any time instant, and
- interleaving of actions if there are more activities to be executed than there are processors available.

The interleaving of actions can be done by taking into account context switching delays for any schedulable action. Further, processors do not idle unless prohibited by synchronization and scheduling constraints. This real-time criterion is imposed at various points in the formal model in order to quantitatively relate the evolution of the program state to the conceptual computation time of the program. The basic constructs of language $\mathcal{L}$ are derived from variants of the concurrency structures used for qualitative concurrency. The important element in the formal model is that available processor resources are made explicit. Further, the semantic model seeks to impose the least restrictions on the scheduler. The model is stronger than the interleaving model and weaker than the maximum parallelism model. Since the model makes processor structure explicit, there is no executional abstraction. This is reasonable for a real-time concurrency model.

These are the central ideas of the limited parallelism model. We are currently investigating the properties of the formal model and the lessons that the formalization can offer for real-time programming. Further we are seeking to extend the model to deal with some open research problems in the area. A semantic model in which processors have local clocks (no assumption of a global clock is made) and the properties of the resulting models has not been reported anywhere in the literature. We are investigating this aspect.

3 Language Concepts

Constructs for specifying priorities are provided in most traditional real-time languages. However no formal justification is given as to the necessity of specifying priorities. The formal semantic model is useful for analying language constructs. One immediate consequence of the formal model of the previous section is that in the absence of proirities for processes, delay commands which in most part are used for asynchronous real-time control loose their essential purpose. Further process cooperation cannot be a priori bounded, thereby defeating the essential purpose in real-time programming. These deficiencies of language concepts are sought to be understood in a formal setting in attached abstract. We come up with several real-time programming concepts. These concepts make the case that *processor holding*, *structured process preemption*, and *simultaneous real-time cooperation* as fundamental for real-time programming. A prototype sketch of the implementation techniques needed for the defined language concepts is currently underway.

Language Concepts for Real-time Concurrency

K.T. Narayana[1] and E. Shade
Whitmore Laboratory
Department of Computer Science
The Pennsylvania State University
University Park, PA 16802, U.S.A.

Abstract

We view the real-time programming problem as one of programming with a priori bounds on the evolution of program states. The bounds themselves are specified in a conceptual time domain. In [ShN88] we formulated a real-time concurrency model under limited processor resources. An analysis based on this formalization reveals that existing concurrency structures are inadequate for real-time programming. We articulate that *processor holding, simultaneous execution of sections of processes* (real-time cooperation), and *structured process preemption* are fundamental language concepts for real-time programming. In the paper, we develop the rationale for the linguistic concepts of **tri-sections, indexed tri-sections** and **indexed-set tri-sections**. We give some examples of programming with these concepts.

1 INTRODUCTION

Atomic actions[Dij68a] are sufficient for concurrent programming under the interleaving models. However real-time programming is one of achieving *a priori* bounds on the evolution of program states. Thus interleaving models are *inadequate*. A quantitative concurrency model is a necessity. For existing concurrency structures, one can provide a quantitative concurrency model[ShN88]. In spite of this formalization, we cannot meet the goals of the real-time programming problem. The central reason is that the semantic model seeks to impose the least implementation restrictions; that is the way it should be. Thus, for meeting the goals of real-time programming, we need additional language concepts which will strengthen the formal model. In the paper, we introduce the language concepts of **tri-sections, indexed tri-sections** and **indexed-set tri-sections**. The concepts formalize the implementation notions of *processor holding*, *simultaneous cooperation*, and *structured process preemption* in a linguistic framework. We articulate that these concepts are fundamental for real-time programming.

1.1 The Limited Parallelism Model

We first highlight some issues with respect to formal models of real-time concurrency. Koymans *et. al.* [KSD85] advocate the *maximal parallelism* model[SaM81] for real-time computing. The model requires that each process in the program be executed on a separate processor. However, in a realistic situation, it is generally the case that there will be fewer processors than processes in the program. Thus the maximal parallelism model is *unrealistic*. A semantic model must therefore cater to limited parallelism.

As a result we need a semantic model for limited parallelism which is stronger than the interleaving model, weaker than the maximal parallelism model, and at the same time does not unnecessarily restrict implementation choices. The work by Shade and Narayana [ShN88] provides a semantic model for real-time concurrency. It considers a small low-level and representative concurrent programming language $\mathcal{L}$. The language uses shared variables for process cooperation, a bus arbitration model of shared memory access, an **await** command for process synchronization, atomic actions for constructing critical sections, and a delay command for real-time control. In the limited parallelism model, the formalization caters to

- maximizing truly concurrent activity at any time instant, and

[1]This work has been partially supported by a research grant from the Office of Naval Research under the grant number N004-89-J-1171.

- interleaving of actions if there are more activities to be executed than there are processors available.

Further, processors do not idle unless prohibited by synchronization and scheduling constraints. This real-time criterion is imposed at various points in the formal model in order to quantitatively relate the evolution of the program state to the conceptual computation time of the program. The basic constructs of language $\mathcal{L}$ are derived from variants of the concurrency structures used for qualitative concurrency. The important element in the formal model is that available processor resources are made explicit. This is reasonable for real-time concurrency. Further, the semantic model seeks to impose the least restrictions on the implementation. The model is stronger than the interleaving model and weaker than the maximal parallelism model.

In spite of such strength, *a priori* bounds on the evolution of program states are not inferrable in the model of [ShN88] due to the potential interleaving of process actions. Thus we need concepts which will strengthen the model further. This paper seeks to formulate the required concepts by first stating the real-time programming problem, and the inadequacy of qualitative concurrency structures to solve the problems.

The paper is organized as follows. In section 2, we characterize the real-time programming problem. In section 3, we reason that qualitative concurrency structures are inadequate for real-time programming. In section 4, we introduce the language concepts of **tri-sections** and **indexed tri-sections** along with their semantic characterization. In section 5, we provide a programming example making use of the introduced language concepts. In the Appendix, we introduce the linguistic concept of **indexed-set tri-sections** for catering to the specification of external nondeterminism. The *full paper* will contain programming examples specifying external nondeterminism and some aspects of the implementation techniques required for the introduced linguistic concepts.

2 What is the Real-Time Programming Problem?

Language concepts and formal models for them go hand-in-hand. If we can formulate what the real-time programming problem is, then with respect to that problem we can address the sufficiency of existing linguistic concepts for concurrency.

We first address the notions of *conceptual time* and *real-time*. *Conceptual time* is a programmer's view of time in his program. *Real-time* is concrete (perhaps measured with respect to an international standard) against which all events of the world are characterized. A programmer, after establishing that his program meets all the conceptual time requirements, relates the notion of conceptual time in his program to real-time by mapping the processor speeds to conceptual time.

For a concurrent program under the interleaving model, we can say that in every computation of the program, if the program satisfies a property P in some state, then *eventually* there exists a state in the computation which will satisfy some property Q. However, in general we cannot characterize a nontrivial bound (even *a posteriori*) on the evolution of the state in which Q is satisfied.

We view the real-time programming problem as one of programmming with *a priori* bounds on the evolution of program states. The bounds themselves are specified in the conceptual time domain. Thus we want to impose an *a priori* bound on the evolution of the state in which Q is satisfied with respect to the state in which P is satisfied. That is, real-time programming is time-dependent programming(!) at least in the conceptual time domain.

This means that a quantitative concurrency model must provide a semantic framework in which an implementation can guarantee that program states evolve consistent with specified bounds. That is, the semantic model proscribes certain computations of the qualitative concurrency model as illegal.

If the chosen concurrency structures are strong enough, then one can reason about the *a priori* bounds from the syntax of the program and its semantic characterization. This approach will lend itself to tractable formal models upon which correctness theories can be built.

3 Inadequacy of Available Concurrency Structures

We now address the problem of whether the available qualitative concurrency structures are sufficient for real-time programming in the light of the fact that the semantic model for quantitative concurrency [ShN88] makes the processor resources explicit. We explain by considering several simple examples.

Example 1 (Independence)

Suppose we are given a program consisting of two independent processes X and Y. Assume that we are given a single processor. Since X and Y are independent, at any instant of time there arc two schedulable actions. Since there is a single processor, we should permit all possible interleavings; this interleaving of actions can take place at each instant of time. Now suppose that we want to impose an *a priori* time bound on the evolution of the program state with respect the computations of a single process. This is impossible in general. The execution of X is dependent on Y in terms of time, since only one processor is available. But X and Y are logically independent by assumption. Thus to characterize a priori bounds, we have to regard the processes as tightly cooperating in spite of their independence. Specifically, let X and Y be as follows.

$$X :: [S_1; S_2]$$
$$Y :: [S_3; S_4]$$

Assume now that property P of the program state holds when control in X is at the statement S_2. Let Q be the property satsfied when control is *after* the statement S_2. We would now like to characterize the a priori bound on the evolution of the program state from the state in which P holds to the state in which Q holds as that designated by the conceptual computation time of S_2. Since X and Y are independent such an a priori bound is not inferrable in [ShN88] due to interleaving.

Since the processor structure is made explicit in the semantic model, we should now see whether it is possible to reprogram X and Y (we want a weak and nontrivial implementation) such that when control in X is at the statement S_2, process Y is blocked and only X is executable. A good reprogramming of Y would then seek to conceptually monitor the control location in X at each instant and block itself once the reprogrammed process Y recognizes that control in X is at S_2. Since one process cannot monitor the control location of another, we need some cooperation between X and Y in order for Y to block itself. Actions pertaining to this cooperation essentially waste the computational resources of the processor for achieving the required blocking. Though theoretically such reprogramming appears possible, in practice the task of reprogramming is horrendous and the amount of wasteful computation is excessive. All that we require from the logical perspective is to say that actions of S_2 cannot be interleaved with actions of any other process. It is here that the qualitative concurrency structures considered in the past are inadequate even under the quantitative model of concurrency[ShN88].

Example 2 (Synchronous Cooperation)

A second deficiency of existing qualitative concurrency structures arises with respect to process cooperation. Assume that we are given three processes X, Y, and Z as follows.

$$X :: [S_1; S_2]$$
$$Y :: [S_3; S_4]$$
$$Z :: [S_5; S_6]$$

Assume that process Z is independent of both X and Y; further cooperation between processes X and Y is synchronous and restricted to the sections of code S_2 and S_4. We would like to *meaningfully* characterize the conceptual execution times of S_2 and S_4. Since they synchronously cooperate, we will only be able to *a priori* characterize the conceptual computation times of S_2 and S_4 in the context of one another. Thus we assume that S_2 and S_4 are simultaneously initiated for execution on two separate processors. In general, relaxation of the notion of simultaneous initiation of S_2 and S_4 does not enable us to characterize *a priori* bounds on their execution.

Because of the logical structure of X, Y, and Z, in the real-time concurrency model of [ShN88] enforcing and hence inferring such bounds is impossible even if the number of processors (namely two) is made explicit. Again we ask, is there a way of reprogramming the process structures X,Y, and Z such that under the real-time model of [ShN88] it is possible to infer the bounds on the execution of S_2 and of S_4 when the two processor structure is made explicit? The critical element in such reprogramming is for the processes X and Y to block themselves when control respectively reaches the statements S_2 and S_4. Having blocked themselves, processes X and Y wait in anticipation that process Z will unblock both of them and then block itself immediately. The assumption here is that when both X and Y are unblocked and further Z is blocked, there will only be the two processes X and Y which are executable. Hence, under the real-time concurrency model of [ShN88], one can infer the *a priori* bounds on the evolution of program state with respect to computations of both X and Y. However, the inherent problem is that process Z must continually monitor the blocking aspect of both X and Y. Thus a good reprogramming, even if it were possible, involves establishing an inherent cooperation regime between X, Y, and Z. Thus we encounter a problem similar to that suggested in the previous example. From a logical perspective, all we require is a mechanism which facilitates the simultaneous execution of S_2 and S_4, such that none of the actions of S_2 and S_4 are interleaved with actions consisting purely of the environment.

Example 3 (Asynchronous Cooperation)

The third example we consider is one of achieving asynchronous real-time cooperation. What do we mean by that? For example, consider the following cooperation of processes.

P_1 :: `compute 10 units of time`; $x :=$ **true**; `compute 10 units of time`; $x :=$ false;

P_2 :: `delay 11`; $y := x$; `compute 10 units of time`; `perform` $f(y)$;

P_3 :: ...

P_1, P_2 and P_3 are processes. The statement `compute 10 units of time` designates some computation which is locally measured to take 10 units of time. The statement `delay 11` takes 11 units of time. Assume that each assignment statement takes unit time. Assume that the precondition for the statement `perform` $f(y)$ requires that y be true. The important element to note is that process P_2 captures the value of x under the full knowledge that P_1 sets x to **true** at time instant 11. In other words, the *a priori* bounds on the evolution of the program state with respect to computations of P_1 and of P_2 have been arrived at locally in spite of the fact that processes P_1 and P_2 are cooperating. Under the maximal parallelism model this asynchronous real-time cooperation is possible. Under the limited parallelism model with 2 processors, since some interleaved actions may correspond to simultaneous execution of P_2 and P_3, the computations may not deliver the required result. Thus delay commands lose their essential purpose in a limited parallelism model.

In all of the examples considered above, the reprogramming problem is one of introducing a cooperation regime on processes which are independent from a logical point of view. Even if we consider a system of processes which are tightly cooperating, subcomputations of those processes can be independent. Thus independence in computations arises dynamically with respect to time. Enforcing *a priori* bounds on specific cooperation aspects of a subset of processes is extremely difficult when the totality of the computation proper exhibits dynamic independence in subcomputations.

It is for these reasons that the available linguistic concepts for real-time concurrency are inadequate. In a formal sense, though strong, the real-time concurrency model of [ShN88] is not strong enough. This inadequacy in strength is brought about by our desire to impose the least implementation restrictions in the semantic model in spite of the fact that the processor structure is made explicit. Thus it is the addition of linguistic concepts that will strengthen the formal model of concurrency.

4 LANGUAGE CONCEPTS

4.1 Tri-Sections

We now introduce the linguistic concept of a *tri-section* $\triangleleft S \triangleright$, with the following semantics.

When the section is executed, no two actions of S will be separated by actions involving the environment of the process to which the section belongs. Assume that every trace of the program consists of the trace of executions of S. Then the semantics of the *tri-section* says that if a_1 and a_2 are any two actions each of unit time duration that occur in sequence in S, then any execution of the program should consist of the actions a_1 and a_2 as consecutive elements in the trace. Viewed differently, by specifying S as a *tri-section* we have restricted the set of of all traces of the program. In terms of the implementation, once a scheduler assigns a processor to a *tri-section*, the processor gets assigned to it until completion of its execution. It is important to note that the scheduler cannot intervene during the execution of the *tri-section*. Note that tri-sections and atomic actions are distinct concepts.

The symbol $\triangleleft$, pronounced *left-tri*, can be interpreted as an executable instruction notifying the scheduler that the rest of the section must be executed without relinquishing the processor. The body of the tri-section must be executed immediately upon the execution of the instruction *left-tri*.

Similarly, the symbol $\triangleright$, pronounced *right-tri*, can be interpreted as an executable instruction which notifies the scheduler that purely environmental actions of the process can be interleaved in any computation thereafter. We further insist that pure environment actions cannot be interleaved between the completion of execution of S and execution of right-tri.

4.2 Labeled and Indexed Tri-Sections

From the examples given earlier, it is clear that the concept of individual tri-sections alone does not solve the problem of achieving real-time cooperation between concurrent sections; we need the additional concept of the simultaneous execution of two tri-sections. As a result we enhance the linguistic primitives in order to specify simultaneously executing tri-sections. Note that the simultaneous *initiation* of sections is critical; otherwise timing may be lost. We assume that all of the tri-sections appearing in each of the processes are *uniquely* labeled. We introduce the language concept of *indexed tri-sections.* The abstract syntax for the indexed tri-section is

$$\ell : \triangleleft_x S \triangleright$$

where the index x on the tri-section is either a constant or a variable designating the set of all labels of tri-sections in addition to itself which should be concurrent. The index set x must consist of at most one label of a tri-section from any process. Since the index x can be a variable, the set of all concurrent tri-sections can be dynamically specified. If x evaluates to $\{\ell\}$, then it is semantically equivalent to $\ell : \triangleleft S \triangleright$.

Designate the symbols $\ell : \triangleleft_x$ with the phrase *left-tri indexed with x*. We can regard the left-tri indexed with x as an executable instruction notifying the scheduler

- that the tri-section labeled with ℓ is enabled, and
- to block the process executing the left-tri indexed with x until a *maximally matching* set of the tri-sections of all processes named in the index set x can be constructed.

(Maximally) Matching Set of Indexed Tri-Sections

A set $\bigcup_{k=1}^{n} \ell_k \cdot \triangleleft_{x_k} S_k \triangleright$ of enabled indexed tri-sections is said to be *matching* iff $x_i = x_j$ for all $1 \leq i, j \leq n$. The set is *maximally matching* if it is matching and $x_1 = \{\ell_1, ..., \ell_n\}$. Note that the formulation is semantic.

Ambiguous Set of Indexed Tri-Sections

A set $\bigcup_{k=1}^{n} \ell_k : \lhd_{x_k} S_k \rhd$ of enabled indexed tri-sections is *ambiguous* iff $x_i \neq x_j$ and $\{\ell_i, \ell_j\} \subseteq x_i \cap x_j$ for some $1 \leq i, j \leq n$.

Abortion of the Program with Respect to Enabled Indexed Tri-Sections

If at any instant of time a set of enabled indexed tri-sections is *ambiguous*, then the program *aborts*. Given an enabled indexed tri-section $\ell : \lhd_x S \rhd$, if the cardinality of the index set x is greater than the number of processors *available*, then the program *aborts*. Note that processor *availability* is a semantic notion.

Progress Specification

Given a *maximally matching* set $\bigcup_{k=1}^{n} \ell_k : \lhd_{x_k} S_k \rhd$ of indexed tri-sections, the set of sections $\bigcup_{k=1}^{n} S_k$ will be initiated for simultaneous execution *immediately* after the maximally matching set was established. Execution of the sections thereafter must be consistent with the semantics of individual tri-sections.

Indexed Tri-Starvation and Indexed Tri-Embrace

An indexed tri-section can starve itself after its enablement if the set of maximally matching indexed tri-sections cannot be constructed in a given computation. Thus it is the programmer's responsibility to prove that there is no starvation with respect to an indexed tri-section.

Similarly the deadly embrace of indexed tri-sections is possible. This can occur if a circularity can be established between any subset of the set of enabled indexed tri-sections. Thus it is again the responsibility of the programmer to prove that there is no deadly embrace of indexed tri-sections.

Notes on Semantics for Indexed Tri-Sections

Given an indexed tri-section $\ell : \lhd_x S \rhd$ in a process, the index set specifies the set of all indexed tri-sections which should agree with x and start executing simultaneously. Since each tri-section in the set must initiate execution simultaneously on a different processor, the mechanism requires a capability for precisely firing the processors at the same time. Such a capability is indeed feasible with the available hardware technology. However, there is a potential for the processors to be out of synchronism by an amount corresponding to *clock-skew*. Clock-skew however is bounded and every formal characterization of real-time concurrency is performed under the assumption that small clock drifts are permissible. Thus we have not made any unrealistic assumptions about the available mechanisms.

4.3 Weakening Tri-Section Semantics

Given a tri-section $\lhd S \rhd$, indexed or not, we have insisted that none of the actions of S can be interleaved with actions purely involving the environment of the process in which the tri-section is defined. Such a strong imposition may interfere with the semantics of the language in which S is specified. Thus we need to be precise about the language in which S is specified. In particular, we require S to be a valid sentence from the low-level concurrent language $\mathcal{L}$ considered in [ShN88]. The language $\mathcal{L}$ as described earlier has primitives for constructing critical sections (or atomic actions).

Since atomic actions can be specified inside a tri-section, a process executing a tri-section can get blocked on an atomic action within the tri-section. Because of blocking, a process may relinquish control of the processor (this is a real-time criterion). This is in contrast to the semantics imposed on the tri-section that no two actions in a tri-section can be interleaved with pure environment actions of the process. Thus we weaken the semantics of tri-sections to admit the possibility of blocking for atomic actions in a tri-section. However, once a process is ready to execute an atomic action in a tri-section, it does so immediately and thereafter the semantics must conform to that of tri-sections.

Experience suggests that in many situations the costs of mutual exclusion are prohibitive in real-time programming[FaP88]. However, the costs can be reduced by precise timing (i.e. asynchronous real-time cooperation) or with the use of busy-waiting (thereby not relinquishing the processor). Since busy-waiting can be specified in the underlying low-level concurrent language $\mathcal{L}$, and the concept of tri-sections does not semantically interfere with the formal characterization of $\mathcal{L}$, such efficiency considerations are left to the programmer.

5 Programming with Tri-Sections

We consider a small real-time example which is typical of a large class of applications found in process control systems.

The system consists of a Task process, a Monitor process, and n Handler processes. The Task process continuously performs some actions and then updates the status of the system. The Monitor process checks the status of the system and records whether the updated status is valid. If there are any problems associated with the status of the system, then it determines which Handler is needed to manage the problem. Once the Monitor process records this, both the Task process and the designated Handler cooperate in real-time to achieve the required corrective effect.

In the program, the statement S designates some finite computation which does not affect any of the shared variables. The variable OK is shared between the Task process, the Monitor process and the set of Handler processes. It records whether the status of the system is valid. Initially, before the system is started, OK is *true*. Only the Monitor process sets OK to *false* upon identifying the nature of the problem in the system state. A variable *handler* is shared between the Task process and the Monitor process. It records a set of labels of Tri-Sections of the Handler processes. The variable is set by the Monitor process which records which Tri-Sections of the Handler processes cooperate in real-time with the Task process. Initially, before the system is started, *handler* equals the empty set. A variable *completed* is shared between the Task process and the set of Handler processes. Initially, before the system is started, *completed* is set to *true*. *completed* is set to *false* by the Tri-Sections in each of the Handler processes. When the Task process has performed a designated action with respect to the problem found, it sets *completed* to *true*. A variable *command* is shared between the Task process and the set of Handler processes. The variable holds an integer depending upon the value of which the Task process takes a corresponding *action*. If the value of *command* is 0, then it will update the status of the system. If *command* has a nonnegative value i, then it performs the corresponding action $action_i$. Initially, before the start of the system, *command* is set to -1. *command* is set by the Handler processes only. Whenever there are problematic symptoms in the status of the system, then the action to be taken is guided by the *Handler* process identifying the symptom. Thus a function $f(i,j)$ yielding a nonzero positive integer value is assigned by the Handler process $Handler_i$ based on the symptom $symptom_j$. The atomic operation **Check_Status** in the Monitor process sets one of the local variables $Status\_ok$ and $problem_i$ to *true* and the others to *false*. The atomic operation **Check_Status** in each of the *Handler* processes sets one of its local variables $symptom_j$ and *no_symptoms* to *true* and the others to *false*.

This briefly explains some intricate aspects of the program and is sufficient to understand the example and the use of tri-sections.

Task::

```
$*[\langle OK_t := OK \rangle;$
    $[\ OK_t \rightarrow S;$ Update_Status
    $[]\neg OK_t \rightarrow \langle x := \{fix\} \cup handler \rangle;$
        $fix : \lhd_x$
            $*[command = 0 \rightarrow$ Update_Status
              $[]_{i=1}^{m} command = i \rightarrow action_i; completed := true;\ await\ (command = 0)]$
            $\rhd]]$
```

```
Monitor::

*[Check_Status;
 [$Status\_ok \rightarrow null$
   $[]_{i=1}^{n} problem_i \rightarrow \langle handler := \{h_i\}\rangle; \langle OK := false\rangle$];
 *[$\langle\neg OK\rangle \rightarrow null$]; $\langle handler := \{\}\rangle$]


$Handler_i$ ::

*[$\langle command := 0\rangle$;
   $h_i : \lhd_{\{fix, h_i\}}$
      *[$\langle\neg OK\rangle \rightarrow command := 0$; Check_Status;
            [$[]_{j=1}^{s} symptom_j \rightarrow completed := false; command := f(i,j); await\ completed$
             $[] no\_symptoms \rightarrow \langle OK := true\rangle; command := -1$]];
      $\rhd$]
```

Notation:

In the program Dijkstra's guarded command notation is employed. A statement $*[A]$ abbreviates the statement $*[true \rightarrow A]$. Statements in **bold** letters designate atomic actions. The command *await* says that the process *busy waits* till the specified boolean condition holds. The statements enclosed between the brackets "$\langle$" and "$\rangle$" designate atomic actions.

Comments on the Example and its Real-Time Response

We can argue that when the program is executed on a configuration of two or more processors, we can guarantee real-time cooperation of the Task process and the relevent Handler process once it is established that there are certain problems in the system status. As long as it can be established that from the initial state every computation leads to a state in which the tri-sections in both the Task process and the relevent Handler process are enabled, then cooperation between two is made possible. Because of interleaving, in general we can say only that the cooperation will *eventually* be initiated. Once it is initiated, however, we can precisely bound its computation time.

We see that when the tri-section *fix* in the Task process is enabled with respect to an enabled tri-section h_i in $Handler_i$, the process Monitor is also ready to execute. By the imposition that enabled maximally matching tri-sections are executed immediately after the establishment of the maximal match, both the Task process and the process $Handler_i$ are scheduled for execution. The Monitor process is effectively preempted — this is important. The use of tri-sections makes it possible to distinguish between the Task process, the Monitor process and the the selected $Handler_i$ process. Thus the concept is richer than a simple synchronization mechanism. It allows us to reason formally about the real-time cooperation of the Task process and the selected $Handler_i$ process.

6 Full Paper Contents

In the full paper, we provide some programming examples using the concept of index-set tri-sections. We further sketch the implementation techniques required for the language concepts.

References

[Dij68a] Dijkstra,E.W, Cooperating Sequential Processes, in *Programming Languages*, F.Genuys (ed.), Academic Press, New York, 1968, pp. 43-112.

[ShN88] Shade,E and K.T.Narayana, *Real-Time Semantics for Shared-variable Concurrency*, Research Report, Department of Computer Science, Pennsylvania State University, University Park, Pa 16802, July 1988.

[FaP88] Faulk,S.R and D.L.Parnas, On Synchronization in Hard Real-Time Systems, *Communications of ACM*, 31, 3, Mar 1988, pp. 274-287.

[KSD85] Koymans, R, *et. al.*, Compositional Denotational Semantics for Real-Time Distributed Computing, *Conference on Logics of Programs*, *LNCS* 193, Springer-Verlag, 1985.

[SaM81] Salwicki,A and T. Müldner, On the Algorithmic Properties of Concurrent Programs, in *LNCS* 193, Springer-Verlag, 1981.

Appendix

A External Nondeterminism and Indexed-Set Tri-Sections

Given an indexed tri-section $\ell : \lhd_x S \rhd$, we have sought to specify that x is a set of labels of tri-sections including its own label. When coupled with the notion of a maximally matching set of indexed tri-sections, the index set x does not offer choice. Since internal nondeterminism in a process is programmable in the low-level concurrent language $\mathcal{L}$ [ShN88] and the notations of tri-sections are built on top of $\mathcal{L}$, the notion of indexed tri-sections considered above is sufficient for programming of internally nondeterministic choice.

In order to cater to specifying external nondeterminism, the structure of the index set x needs to be enhanced. Instead of requiring x to be a set of labels of tri-sections, we require that x be specified as a set consisting of sets of labels. We call such a tri-section an *indexed-set tri-section*. This phrase will distinguish our vocabulary with respect to an *indexed tri-section*. We consider that an indexed tri-section $\ell : \lhd_x S \rhd$ to be equivalent to the indexed-set tri-section $\ell : \lhd_{\{x\}} S \rhd$. We formulate the restrictions on x in an indexed-set tri-section.

Given an indexed-set tri-section $\ell : \lhd_X S \rhd$, where $X = \{b_1, \ldots, b_m\}$, $m \geq 1$, it must be case that for all $1 \leq i \leq m$, $\ell \in b_i$ and b_i consists of at most one label of a tri-section from any process. We interpret $\lhd_X$, the left-tri indexed with X, as an executable instruction notifying the scheduler

- that the tri-section labeled with ℓ is enabled, and
- to block the process executing the left-tri indexed with X until the tri-sections of all processes named in some $b_i \in X$ are enabled such that there exists a maximal match.

Matching Set of Indexed-Set Tri-Sections

A set $\bigcup_{k=1}^{n} \ell_k : \lhd_{X_k} S_k \rhd$ of enabled indexed-set tri-sections is *matching* iff $\bigcap_{k=1}^{n} X_k \neq \emptyset$. The set is *maximally matching* if $\bigcup_{i=1}^{n} \ell_i \subseteq \bigcap_{j=1}^{n} X_j$.

Ambiguous Indexed-Set Tri-Sections

Given a pair of indexed-set tri-sections $\{\ell_1 : \lhd_{X_1} S_1 \rhd, \ell_2 : \lhd_{X_2} S_2 \rhd\}$, where $X_1 = \{b_1, \ldots, b_m\}$ and $X_2 = \{c_1, \ldots, c_n\}$, $m, n \geq 1$, the pair is *ambiguous* iff $b_i \neq c_j$ and $\{\ell_1, \ell_2\} \subseteq b_i \cap c_j$ for all $1 \leq i \leq m$ and $1 \leq j \leq n$. A *set* of indexed-set tri-sections is *ambiguous* if and only if there exists an *ambiguous pair* in the set.

Abortion of the program with respect to enabled indexed-set tri-sections

If at any instant of time, a set of enabled indexed-set tri-sections is *ambiguous*, then the program *aborts*. If a maximally matching set of indexed-set tri-sections has cardinality greater than the number of available processors, then the program *aborts*.

Progress Specification

Given a *maximally matching* set of indexed-set tri-sections $\bigcup_{k=1}^{n} \ell_k : \lhd_{X_k} S_k \rhd$, the set of sections $\bigcup_{k=1}^{n} S_k$ will be simultaneously initiated for immediate execution at the time instant in which the maximally matching set was established. Execution of the sections must be consistent with the semantics of individual tri-sections.

If at any instant more than one maximally matching set can be established, then the choice is nondeterministic.