Syntax and Semantics of PRETSEL–A Specification Language for Parallel Real-Time Systems*

Alok Choudhary
ECE Department
Syracuse University
Syracuse, NY 13244, USA

Vijay Gehlot
CIS Department
University of Pennsylvania
Philadelphia, PA 19104, USA

Bhagirath Narahari
EE & CS Department
The George Washington University
Washington, DC 20052, USA

Abstract

For many real-time applications (e.g. Command, Control, and Communications), parallel computers offer a natural computing platform. However, very little attention has been paid to the specification requirements of real-time systems implemented on parallel machines. Towards this end, we propose a specification language PRETSEL (Parallel REal-Time SpEcification Language). The PRETSEL specification language is based on a traditional two-level view of 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 way (task-parallelism). Furthermore, the PRETSEL language allows for the specification of performance requirements. PRETSEL is currently being evaluated for real-time avionics applicaitons. In this paper we describe the sysntax and operational semantics of PRETSEL and establish results relating the functional and the temporal behaviors.

1 Introduction

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. 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.

We believe that complex software systems and especially real-time systems can be made truly robust and reliable if powerful specification and analysis techniques are made available to software developers and maintainers. In recent years there has been a significant progress in the development of formal models for real-time computing. These include timed automata [1], Timed Petri Nets [4], Timed CSP [5], Z and RTL [6], Timed Process Algebra [2, 8], ACSR [12], Timed CCS [15], Temporal Logic [9, 10]. The model we are proposing here differs from these in the following respects. 1) There are no instantaneous actions in our model—all actions consume time. 2) The semantics of parallel operator is *must synchronize* when complementary actions are involved as opposed to *may synchronize* of CCS-like languages. Furthermore, this seems to obviate the need for restriction operator in our language. 3) The timing operators are of more general nature assigning a range of time values to actions instead of an exact duration. This makes our model more close to reality since, for example, in practice execution time of actions depends on various factor and will vary. Hence assigning a range of values or a bound to an action is more meaningful than stipulating it to be an exact value. 4) The choice operator in language is biased. It favors the component which may finish earlier. This allows us to define, for example, multiple versions of the same task for, say, different mappings or machines, etc. 5) Our language provides an abstraction operator to abstract system dependent features. This appears useful in defining *performance polymorphism* in the sense of [11]. 6) Finally, and most importantly, the existing models do not adequately address the specification requirements for realistic real-time system software on parallel computers. Our language provides the traditional constructs to define a data-parallel algorithm. It should be emphasized

*This research is supported by contract No. F3-6-2-94-C-0073 from the Rome Laboratory.

that parallel computing adds complexities to a real-time system which are normally absent in uniprocessor systems. In real-time systems, performance correctness (i.e., meeting deadlines etc.) is as important as functional correctness. However, performance of a parallel algorithm on a parallel computer, depends on a number of architectural and algorithmic properties, such as number of processors, communication, scalability of algorithms, overhead of scheduling parallelism and synchronization. These factors do not arise in sequential processes, but must be taken into consideration by any specification model for parallel real-time systems. The model must also provide features to recognize changes in the environment (such as change in input data rate) and thereby respond by reallocating resources to meet the timing requirements. The necessity of specifying some of the parameters described above is illustrated by the following examples.

Typically the speedup per processor of a parallel algorithm, also called the efficiency, decreases with increasing number of processors (due to more communication) and also depends on the input characteristics. These scalability parameters must be included in the program specification. Different parallel algorithms for the same computation can have different efficiency functions. For example, we may have two algorithms for sorting –one which works well for large number of processors and the other that is tailored for coarse grained parallelism. As the scalability parameters vary, the specific of algorithm to use to meet the performance requirements may change. Thus, there is a need for provision of multiple versions/algorithms to carry out a given computation and a specification model must capture the scenario described above by specifying these multiple versions and the precise metrics used for selecting each version. Depending on the state of the system and the performance requirement the appropriate algorithm is selected.

As another example, consider the continuous processing of data arriving at real-time rates (this could be considered as a periodic task, since the same computations must be performed for different data sets). A sensor task collects data at some rate and sends it to a task that processes the data. For example images received and sent to an image processing algorithm. When there is a bursty I/O, *i.e.,* data arrives at a more rapid rate, the amount of data to be processed may change drastically and thus the parallel algorithm may no longer meet its time deadlines. The system must detect this change and a *remapping* process must be invoked to determine a parallel algorithm and additional processors that have to be used to meet the deadline. The overhead of this process must also be taken into account. This entire process of resource allocation must be specifiable and verifiable by a specification model.

The above examples illustrate why the the specification must model the performance (of the algorithm) as a function of the system characteristics. In addition, the communication requirements in parallel algorithms are far more complex than sequential or real-time systems. In particular, a number of communication primitives must be provided by the system and included in the specification. The same holds for synchronization mechanisms.

In this paper we propose a formal specification language called PRETSEL—Parallel REal Time SpEcification Language— for real-time systems implemented on parallel machines. The computation model of PRETSEL is based on the view that A parallel real-time computation is, in general, a collection of interacting processes, each of which can be a parallel algorithm. At this stage in our research, we consider the case where each process is a data parallel algorithm. Modelling parallel computations in this manner naturally leads to a two-level specification model. At level 2 we provide constructs for specifying data parallel algorithms, and at level 1 we provide constructs to combine such tasks in a variety of ways. Thus, for example parallelism occurs at two levels - within a task (data parallelism) and among tasks (functional or task parallelism). A level 2 process 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 rest of this paper is organized as follows. The next section introduces the syntax of PRETSEL. This is followed by semantics of PRETSEL in Section 3. We finally conclude in Section 4.

2 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

Proceedings of the 1996 Hawaii International Conference on System Sciences (HICSS-29)
1060-3425/96 $10.00 © 1996 IEEE

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 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, we stipulate a set of action symbols *Act*.Our time domain $\mathcal{T}$ is the set of natural numbers plus infinity, that is, $\mathcal{T} = \mathcal{N} \cup \{\infty\}$. Since all our 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, we 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 $\delta(a) = t$. We 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. We also assume that Act_i and Act_e can be partitioned into two equinumerous sets with a complementation bijection, denoted $\bar{}$, 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 1 where *min_time* and *max_time* range over the time domain $\mathcal{T}$. The syntax of Level 2 tasks is shown in Figure 2.

We 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 various operators at level 1 is as follows. The parallel composition $P_1 \parallel P_2$ denotes a process where two components P_1 and P_2 proceed in time independently of each other except for synchronizations. 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 us to specify different versions of an algorithm to perform the same computation, such that the algorithm that meets the deadlines will be selected.

Currently, we have three types of timing constraints (or specifications): (1) $\Phi\, min\_time$, (2) $\Omega\, max\_time$, and (3) $\Delta\,[t_1, t_2]$. The first one 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 at duration interval, between t_1 and t_2, for the computation. Note that we can define exact timing 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 when we present the temporal rules in the next section). The periodic operator Π can be used to define a periodic process at level 1. For example, $\Pi^t P$ is a process that does P every t time units.

Now consider the level 2 syntax which specifies the 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 our 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 $(\mathbf{add}, \Omega 2)$ 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 our model of computation. The computations can be arithmetic operations. The internal communications would include communication primitives such as the send-receive primitive, barrier synchronization, broadcast, etc. The external communications would include communication needed with other tasks to exchange data, for I/O activities, and pure synchronization with other tasks. 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 [13]. Similar time scoping constructs and actions that consume time have been used in [7]. It should be noted that the oper-

Proceedings of the 1996 Hawaii International Conference on System Sciences (HICSS-29)
1060-3425/96 $10.00 © 1996 IEEE

$$
\begin{array}{rcl}
\langle process \rangle & ::= & \langle task \rangle \\
& | & \langle process \rangle \parallel \langle process \rangle \\
& | & \langle process \rangle + \langle process \rangle \\
& | & \langle process \rangle \Rightarrow \langle process \rangle \\
& | & \langle time\_spec \rangle : \langle process \rangle \\
& | & \Pi^{\langle time\_expr \rangle} \langle process \rangle \\
\\
\langle time\_spec \rangle & ::= & \Omega\, max\_time \\
& | & \Phi\, min\_time \\
& | & \Delta\, [\, min\_time\, ,\, max\_time\,]
\end{array}
$$

Figure 1: Level 1 Syntax

ator & is similar to the binary case of **forall** of [3]. Since such **foralls** are so pervasive in parallel programming that we define the following derived operator:

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

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

3 Semantics of PRETSEL

The above discussion provided an informal view of the semantics of PRETSEL, and we now discuss the operational semantic rules for PRETSEL. The operational meaning of PRETSEL operators may depend on *temporal correctness* of processes and tasks. To capture temporal correctness we define a set of *temporal rules*. For sake of brevity and simplicity, we shall restrict ourselves only to Ω constraints here. Figure 3 and Figure 4 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 we can infer the transition below the line. 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*.

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 language, the temporal rules may be thought of assigning temporal information to expressions. In the case where we restrict ourselves to Ω, this 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 $P_1 \parallel P_2$ completes when both its components have completed and hence the time taken is the maximum of the time taken by either component. Rule (2) says that the choice composite of two processes $P_1 + P_2$ finishes as soon as one of them is done. Rule (3) says 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) says 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) says that the time to complete an *if* operation is the maximum of the time taken to complete the consequent and the alternative. Since we cannot a priori determine the

Proceedings of the 1996 Hawaii International Conference on System Sciences (HICSS-29)
1060-3425/96 $10.00 © 1996 IEEE

$$
\begin{array}{rcl}
\langle task \rangle & ::= & \langle basic\_task \rangle \mid [\langle sys\_specs \rangle]\langle basic\_task \rangle \\
\langle sys\_specs \rangle & ::= & \langle sys\_spec \rangle, \langle sys\_specs \rangle \\
\langle sys\_spec \rangle & ::= & num\_proc \mid input\_spec \mid exec\_time\_spec \mid map\_spec \mid arch\_spec \\
\langle basic\_task \rangle & ::= & \langle action \rangle \\
 & \mid & \langle basic\_task \rangle \ \& \ \langle basic\_task \rangle \\
 & \mid & \langle basic\_task \rangle \ ; \ \langle basic\_task \rangle \\
 & \mid & if \ \langle bool\_expr \rangle \langle basic\_task \rangle \langle basic\_task \rangle \\
 & \mid & while \ \langle bool\_expr \rangle \langle basic\_task \rangle \\
 & \mid & every \ \langle time\_expr \rangle \langle basic\_task \rangle \\
 & \mid & within \ \langle time\_expr \rangle \langle basic\_task \rangle
\end{array}
$$

Figure 2: Level 2 Syntax

number of iterations, according to rule (10) the maximum time taken by a *while* construct is bounded by ∞. Rule (11) is analogous to rule (5) for processes. Rule (12) says that the temporal scope of a task must be compatible with the timing requirement of its body. Rule (13) requires a bit explanation. We 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 says 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 having a possibly different performance characteristics. This has been termed *performance polymorphism* in [11]. As an example, consider a simple task consisting of just one action $T =$ **add**. This action may take different time to execute depending on the underlying architecture. To capture this variation, we may abstract away this information and define $T' = [arch\_spec]T$. The task T' may now be instantiated with different architecture specifications that will in turn set the execution time of the **add** operation as given by the mapping δ.

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 $P =$ **add ; send**. Let us further suppose that on a given machine we know how long the **add** operation is going to take, say, $\delta(\mathbf{add}) = 2$ but we do not know how long the **send** operation takes. Furthermore let us suppose that we want P to finish in 10 time units, that is, $\Omega 10 : P$ is what we want. Using rules(4), (6), and (7) it is easily deduced that the send operation must be completed within 8 units of time. This is depicted in the deduction tree below:

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

The desired deduction follows in trying to build (backwards) a proof-tree of the goal $(\Omega 10 :$ **add ; send**$) : 10$. From the application of rule 4, it can be deduced that the desired goal is provable if we can establish that (**add ; 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 $x \leq 8$. This kind of information can be statically deduced and can be used at compile time for scheduling etc.

Proceedings of the 1996 Hawaii International Conference on System Sciences (HICSS-29)
1060-3425/96 $10.00 © 1996 IEEE

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

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

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

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

$$\frac{P : t \qquad t \leq t'}{\Pi^{t'} P : \infty} \tag{5}$$

Figure 3: Level 1 Temporal Rules

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

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

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

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

$$\frac{T : t \qquad b : t_1}{while \; b \; T : \infty} \tag{10}$$

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

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

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

Figure 4: Level 2 Temporal Rules

Proceedings of the 1996 Hawaii International Conference on System Sciences (HICSS-29)
1060-3425/96 $10.00 © 1996 IEEE

Next we turn our attention to operational rules. The operational rules for level 1 and level 2 are contained in Figure 5 and Figure 6, respectively. The operational rules are transition based. In giving these rules, we let a 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 we are not separating time from action, we do not need to define two separate transition relations as has been done in [15]; rather our approach is similar to that of [12] 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 we call the *must synchronize* semantics of $\|$ which differs from what may be called the *may synchronize* semantics of CCS [14]. Because of this, CCS provides another operator called *restriction* to force synchronization. Our choice of must semantics then obviates the need for a restriction-like operator—at least for synchronization purposes.

We should point out that PRETSEL in fact provides a variety of communication primitives each with its own set of rules. Thus, for example, for non-blocking `send` and complementary `recv` the following rule would be applicable:

$$\frac{P_1 \xrightarrow{\texttt{send(v)}} P_1' \qquad P_2 \xrightarrow{\texttt{recv(x)}} P_2'}{P_1 \| P_2 \xrightarrow{\tau} P_1' \| P_2'[v/x]}$$

It should also 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, 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) says 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). The 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.

In the above we described temporal rules to capture the timing requirements of a given process or a task and we 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 $P : t$. If $P \xrightarrow{a_1} P_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n}$ done then $\sum_{i=1}^{n} \delta(a_i) \leq t$.*

Also our 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 examples, that the processes are time correct. So, the question is whether we can '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

Proceedings of the 1996 Hawaii International Conference on System Sciences (HICSS-29)
1060-3425/96 $10.00 © 1996 IEEE

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

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

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

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

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

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

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

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

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

Figure 5: Level 1 Functional Rules

Proceedings of the 1996 Hawaii International Conference on System Sciences (HICSS-29)
1060-3425/96 $10.00 © 1996 IEEE

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

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

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

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

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

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

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

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

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

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

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

Figure 6: Level 2 Functional Rules

answer relies on an erasure mapping that erases all the timing information and what we are left with is only the functional part. Details of this erasure mapping $\mathcal{E}$ will be presented elsewhere. Just to give an idea of $\mathcal{E}$, we fist 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.

4 Conclusion and Future Work

This paper discussed the problem of formal specification of real-time systems implemented on a parallel machine. Towards this end we proposed the PRETSEL specification language which allows specification of functional, timing, and performance requirements. PRETSEL takes a two level approach and explicitly addresses parallelism issues at a higher and more realistic level, and reflects the computation model used commonly in the parallel processing community. We described the syntax and formal semantics of PRETSEL. In defining the formal semantics, two classes of rules were given for expressions at each level. These were classified as *temporal rules* and *operational rules*. Temporal rules assign temporal attribute to expressions

Proceedings of the 1996 Hawaii International Conference on System Sciences (HICSS-29)
1060-3425/96 $10.00 © 1996 IEEE

much the same as the typing rules of a typed language assigns types to programs. Furthermore, just as (most) typing is a static property, so is the temporal attribute in our case. Also, just as type information can be utilized in a useful way during compilation, we illustrated by means of a simple example that the temporal attributes given by temporal rules can be used for scheduling, etc. The operational rules capture the 'execution' of processes. Our operational rules combine both the functional behavior and the temporal behavior into one relation. We also establish that the temporal attribute assigned by temporal rules is consistent with the temporal behavior of the process involved. This was summarized in proposition 1. We also showed that pure functional behavior, without the overhead of temporal constraint checking, can be obtained from our operational rules and that the resulting functional behavior would be correct in the sense that if the timed process terminates than so does its purely functional counterpart. This was contained in proposition 2.

Our current work is towards evaluating the expressivity and naturalness of PRETSEL. Towards this end, we are using PRETSEL for specifying large real-life systems [16]. This exercise may force us to modify/extend PRETSEL. One possible extension of PRETSEL would be to include exception handling. There has already been significant work done in this direction by other researchers in this area—most notably the one reported in [12]—and we hope to capitalize on it. Another direction that we would like to pursue is the design and development of a (semi) automated verification/synthesis tool based on PRETSEL.

References

[1] R. Alur and D. Dill. Automata for modeling real-time systems, *Proc. of 17th ICALP, LNCS 443*, pp. 322–335. Springer Verlag, 1990.

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

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

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

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

[6] 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, pp. 393–408, Springer-Verlag 1991.

[7] A. N. Fredette and R. Cleaveland, RTSL: A language for real-time schedulability analysis, *Proc. Real-Time Systems Symposium*, pp. 274–283, Raleigh-Durham, North Carolina, December 1993.

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

[9] T. Henzinger, Z. Manna, and A. Pnueli, Temporal proof-methodologies for real-time systems, *Proc. ACM Principles of Programming Languages*, 1991.

[10] J. Hooman, *Specification and Compositional Verification of Real-Time Systems*, LNCS Vol 558, Springer-Verlag, 1991.

[11] 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.

[12] 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.

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

[14] R. Milner. *Communication and Concurrency* Prentice-Hall, 1989.

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

[16] J. Wenner, Specification of Real-Time Systems on Parallel Computers: Sonar Beamformer Example, Master's Project, ECE Department, Syracuse University, January 1995.

Proceedings of the 1996 Hawaii International Conference on System Sciences (HICSS-29)
1060-3425/96 $10.00 © 1996 IEEE