Specifying Functional and Timing Behavior for Real-Time Applications

Mario R. Barbacci[1,2] and Jeannette M. Wing[2]
Carnegie Mellon University
Pittsburgh, PA 15213, U.S.A.

Abstract

We present a notation and a methodology for specifying the functional and timing behavior of real-time applications for a heterogeneous machine. In our methodology we build upon well-defined, though isolated, pieces of previous work: Larch and Real Time Logic. In our notation, we strive to keep separate the functional specification from the timing specification so that a task's functionality can be understood independently of its timing behavior. We show that while there is a clean separation of concerns between these two specifications, the semantics of both pieces as well as their combination are simple.

1 Problem Context

Many computation-intensive, real-time applications require efficient concurrent execution of multiple *tasks* devoted to specific pieces of the application. Typical tasks include sensor data collection, obstacle recognition, and global path planning in applications such as robotics and vehicular control. Since the speed and throughput required of each task may vary, these applications can best exploit a computing environment consisting of multiple special and general purpose processors that are logically, though not necessarily physically, loosely connected. We call this environment a *heterogeneous machine*.

During execution time, *processes*, which are instances of tasks, run on possibly separate processors, and communicate with each other by sending messages of different types. Since the patterns of communication can vary over time, and the speed of the individual processors can vary over a wide range, additional hardware resources, in the form of switching networks and data buffers are required in the physical heterogeneous machine. Logically, *queues* are used to buffer data; processes dequeue data on queues attached to input ports and enqueue data from queues attached to output ports.

The application developer is responsible for prescribing a way to manage all of these

---

[1]Software Engineering Institute, [2]Department of Computer Science

This research is carried out jointly by the Software Engineering Institute, a Federally Funded Research and Development Center, sponsored by the Department of Defense, and by the Department of Computer Science, sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory Under Contract F33615-84-K-1520. Additional support for J.M. Wing was provided in part by the National Science Foundation under grant DMC-8519254.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Software Engineering Institute, Carnegie-Mellon University, the National Science Foundation, the Department of Defense or the US Government.

resources. We call this prescription a *task-level application description.* It describes the tasks to be executed, the assignment of processes to processors, the data paths between the processors, and the intermediate queues required to store the data as it moves from source to destination processes. A *task-level description language* is a notation in which to write these application descriptions.

We are using the term "description language" rather than "programming language" to emphasize that a task-level application description is not translated into object code in some kind of executable "machine language." Rather, it is to be understood as a description of the structure and behavior of a logical machine, that will be synthesized into resource allocation and scheduling directives. These directives are to be interpreted by a combination of software, firmware, and hardware in a heterogeneous machine.

```
task task-name
  ports                          -- Used for communication between a process and a queue
    port-declarations
  signals              -- Used for communication between a user process and the scheduler
    signal-declarations
  behavior             -- A description of the functional and timing behavior of the task
    requires predicate
    ensures predicate
    timing timing expression
  attributes                                          -- Additional properties of the task
    attribute-value-pairs
  structure          -- A process-queue graph describing the internal structure of a task
    process-declarations
    queue-declarations
    reconfiguration-statements
end task-name
```

Figure 1: A Template for Task Descriptions

We have an initial design of such a description language [3], a compiler for it, and a simulator that takes task descriptions as input. A task description (see Figure 1) contains information about four aspects of a task: (1) its interface to other tasks (**ports**) and to the scheduler (**signals**), (2) its functional and timing **behavior**, (3) its **attributes**, and (4) its internal **structure**, thereby allowing for hierarchical task descriptions. Reference [3] contains a more complete explanation of these and other features of the language. In this paper we focus on only one aspect: the information appearing in the **behavior** part of a task description.

2 Contributions

Formal specifications have been used successfully for specifying the functional behavior of software systems, e.g., individual program modules and abstract data types. These specifications have traditionally been used to verify a program's correctness ("is the right answer computed?"). Often, however, one is interested in not only the functional correctness of a system but also other properties, such as reliability, performance, security, and real-time behavior. Less work has focused on formally specifying these other properties of software systems, let alone their interactions with each other.

To our knowledge no work has addressed the formal integration of the formal specification of functional and timing behavior of software. The main contribution of this paper is exactly this integration of functional and timing specifications as embodied in our task description language.

We combine two separate formalisms: an axiomatic specification language, Larch [15, 16], used to specify functional behavior, and an event expression language used to specify timing behavior. Both are mapped to the same underlying logic, typed first-order predicate logic, so that their combination has a formal semantics.

Two significant aspects of our work are as follows:

- Since the formal semantics is relatively simple (first-order logic), not only can people easily understand our specifications but the specifications themselves can easily be subject to machine analysis.
- We build upon previous well defined and isolated pieces of research and combine them in a meaningful way. Their combination is applied in a context (heterogeneous machines) that itself is of growing interest to those involved in parallel architectures and languages.

3 Introduction to Larch

Before we describe the functional and timing specifications of a task, we give a brief introduction to Larch. We are keeping this introduction to Larch very short. The reader is encouraged to consult the appropriate references in the bibliography.

Larch uses a two-tiered approach to specifying program modules: a *trait* defines state-independent properties, and an *interface* specification defines state-dependent properties of a program. A trait is written in the Larch Shared Language (LSL), and it provides the assertion language used to express and define the meaning of the predicates of an interface specification.

```
QVals: trait
  introduces
    empty: → Q
    insert: Q, E → Q
    first: Q → E
    rest: Q → Q
    isEmpty: Q → Boolean
    isIn: Q, E → Boolean
  constrains Q so that
    Q generated by [ empty, insert ]
    for all q: Q, e, e1: E
        first(insert(empty), e)) = e
        first(insert(q, e)) = if isEmpty(q) then e else first(q)
        rest(insert(q, e)) = if isEmpty(q) then empty else insert(rest(q), e)
        isEmpty(empty) = true
        isEmpty(insert(q, e)) = false
        isIn(empty, e) = false
        isIn(insert(q, e), e1) = (e = e1) | isIn(q, e1)
```

a. A Trait for Queue Values

```
Enqueue = operation (q: queue, e: element)
    ensures q_post = insert(q, e)
Dequeue = operation (q: queue) returns (e: element)
    requires ~isEmpty(q)
    ensures q_post = rest(q) & e = first(q)
```

b. Interfaces for Queue Operations

Figure 2: A Larch Two-Tiered Specification for Queues

For a program module, such as a procedure, a Larch interface specification is written in a Larch Interface Language (LIL) and contains predicates about the states before and after the execution of the procedure. The Larch Interface Language to be used is specific to the programming language in which the procedure is written (e.g., C, Common Lisp, Ada, etc.). For this paper we will use a relatively simple interface language, such as would be defined for an Algol-like language.

Figure 2 depicts a Larch (two-tiered) specification of queues with Enqueue and Dequeue operations. The top part of the specification (Figure 2.a) is a trait written in LSL used to describe values of queues. A trait is akin to an algebraic specification (see Section 7 on Related Work). A set of operators and their signatures following **introduces** defines a vocabulary of terms to denote values of a type. For example, empty and insert(empty, 5) denote two different queue values. The set of equations following the **constrains** clause defines a meaning for the terms; more precisely, an equivalence relation on the terms, and hence on the values they denote. For example, from the above trait, one could prove that first(rest(insert(insert(empty, 5), 6))) = 6.

The bottom part of the specification (Figure 2.b) contains two interfaces written in our "generic" Larch interface language. They describe the functional behavior of two queue operations, Enqueue and Dequeue (queue operation names are used to write timing expressions, which are described later in this paper). A **requires** is a pre-condition on the state of an operation's input data that must be true upon operation invocation; an **ensures** is a post-condition on the state of an operation's input and output data that is guaranteed to be true upon operation termination. An omitted predicate is taken to be **true**. The specification for Dequeue states that Dequeue must be called with a non-empty queue and that it modifies the original queue by removing its first element and returning it.

4 Behavioral Information

The behavioral information in a task description is divided into two parts: a functional specification and a timing specification. In the next two subsections we describe informally the syntax and meaning of these two specifications. Section 5 gives the formal meaning, and in particular, the meaning of the combination of functional and timing specifications.

4.1 Functional Specifications

The functional information of a task description (see Figure 1) describes the behavior of the task in terms of predicates about the data in the queues, before and after each execution of the task. It consists of a **requires** clause and an **ensures** clause, together constituting a simple Larch interface specification. LSL is used as the assertion language in the predicates of these clauses.

A **requires** clause states what is required to be true of the data coming through the input ports; an **ensures** clause states what is guaranteed to be true of the data going out through the output ports. If one were to view each cycle of a task as one execution of a procedure, the **requires** and **ensures** are exactly the pre- and post-conditions on the functionality of that cycle.

A task implementation must satisfy the predicates, R and E, of the **requires** and **ensures** clauses. A task implementation is simply a program written in some programming language, e.g., C, Common Lisp, or Ada. Using Hoare-like notation, an implementation, Prog, *satisfies* the (functional) specification if:

$$\{R\}\ \text{Prog}\ \{E\}$$

It is up to the task implementor to show that a task implementation satisfies the functional specification as given by the **requires** and **ensures** clauses. This verification can be done formally — standard verification techniques can be used ([17, 18]) and some mechanical tools are available to aid this process ([13, 22, 24]). We defer to Section 5.2 for the definition of the meaning of the predicates in the presence of timing information.

```
task multiply
  ports
    in1, in2: in matrix;
    out1: out matrix;
  behavior
      requires  number_of_rows(first(in1)) = number_of_cols(first(in2));
      ensures   out1post = insert(out1, first(in1) * first(in2));
end multiply;
```

Figure 3: The Functionality of a Matrix Multiplication Task

For example, consider a matrix multiplication task (Figure 3) that takes input matrices from two queues and outputs the result matrix on an output queue. The data traveling through these ports are of type matrix. Matrix values are specified using LSL just as for queue values, so "number_of_rows," "number_of_cols" and "*" would be defined in a trait about matrix values. The **requires** clause states that the task implementor may assume that the number of rows of the matrix entering through the port in1 equals the number of columns of the matrix entering through in2. The **ensures** clause states that the result of multiplying the two input matrices in one cycle is output to the queue attached to the output port.

4.2 Timing Specifications

The timing information describes the behavior of the task in terms of the operations that it performs on the queues attached to its input and output ports; this is the behavior of the task seen from the outside. A timing expression is a regular expression built from concurrent and sequential queue operations, with optional conditional expressions or guards that control when a subexpression is to be executed. Timing expressions are similar to a number of formalisms derived from Path Expressions [7].

The simplest timing expression is the name of a queue operation on the queue attached to a specific port, e.g., in1.Dequeue or OutPort.Enqueue. The duration of a queue operation or the delay between two operations is described by a time window, denoted by a pair of time values, $[T_{min}, T_{max}]$, defining the boundaries of the interval. The time window associated with a queue operation describes the minimum and maximum time needed to perform the operation (in1.Dequeue[15,25]) Intervals of time between queue operations are denoted by a Delay "operation" whose time window describes the minimum and maximum time consumed by the process in between queue operations.

A composite timing expression denotes the sequential and/or concurrent execution of operations on queues. Sequential composition is denoted by a space between operations; parallel composition is denoted by a "||" between operations. For example,

loop (in1.Dequeue[10,15] || in2.Dequeue) delay[*,30] out1.Enqueue

is a sequential timing expression that specifies two parallel Dequeue operations on the queues attached to the input ports in1 and in2, followed after some delay by an Enqueue on the queue

attached to the output port out1. The Delay lasts some undetermined amount of time less than 30 seconds. The Dequeue operation on port in1 takes between 10 and 15 seconds to complete. The other two operations take some implementation dependent default time to complete. The keyword **loop** denotes a cyclic or repeating task.

1. The task is executed some integral number of times:

   repeat *integer* => *expression*

2. The task is allowed to start during some time interval:

   during *timewindow* => *expression*

3. The task is allowed to start no earlier than some time value:

   after *timevalue* => *expression*

4. The is allowed to start no later than some time value:

   before *timevalue* => *expression*

5. The task is allowed to start only if some predicate on the state of the input queues or the current time is true:

   when *predicate* => *expression*

Table 1: Timing Expression Guards

An optional guard in a timing expression specifies a restriction on the execution of the associated timing expression, as shown in Table 1.

```
task multiply
  ports
    in1, in2: in matrix;
    out1: out matrix;
  behavior
      requires number_of_rows(first(in1)) = number_of_cols(first(in2));
      ensures out1post = insert(out1, first(in1) * first(in2));
      timing when (~isEmpty(in1) and ~isEmpty(in2)) =>
                  ((in1.Dequeue || in2.Dequeue) delay[10,15] out1.Enqueue);
end multiply
```

Figure 4: The Timing of a Matrix Multiplication Task

For example, consider a revised matrix multiplication task (Figure 4). The **timing** clause states that the task does not start executing until both input queues contain data. Once that condition is satisfied, the task will remove its input data from both input queues concurrently (the Dequeue operations), will operate on the data for between 10 and 15 seconds (this "computation" time is lumped together under the delay operation), and finally will enqueue some output in the output queue. Notice another use of LSL in our specifications: the **when** condition places a constraint on the state of the queues (not on the state of the data in the queues). We use the trait from Section 3 to define the assertion language for predicates in a **when** guard.

5 Formal Meaning of Functional and Timing Specifications

We use Jahanian and Mok's Real-Time Logic (RTL) [19] to give meaning to our timing expressions. Furthermore, we use their logic to give meaning to the combination of our functional and timing specifications. We use four of their notational conventions:

Syntax	*Meaning*
↑A	The start of an operation ("action" in RTL's terminology).
↓A	The end of an operation.
@(E, i)	The time of the i^{th} occurrence of event E, where events in our context are the start of an operation or the end of an operation. @ is an occurrence function that captures the notion of real-time.
P(t1, t2)	The interval of time during which the predicate P holds. P holds before or at t1, from t1 to t2, and at or after t2. If t1 and t2 are identical, then P holds at an interval around t1. For brevity, we will use P(t) when t1=t2 (i.e., "P holds around time t").

5.1 Assigning Meaning to Timing Specifications

In this section we describe the meaning of our timing specifications in terms of RTL logic. In the following discussion, we assume E, E1, and E2 are arbitrary timing expressions; A, A1, and A2 are operations; t1 and t2 are times (absolute or relative); a1 and a2 are absolute times; r1 and r2 are relative times: and W is a predicate of a **when** guard.

1. For any queue operation A, and for some implementation defined time window [T1,T2], the following axiom expresses the (default) duration of the operation:

$$\forall i\ [\ T1 \le @(\downarrow A,i) - @(\uparrow A,i) \le T2\ ]$$

2. For any queue operation A[t1,t2], with a duration defined by the time window [t1,t2], the following axiom expresses the duration of the operation:

$$\forall i\ [\ t1 \le @(\downarrow A,i) - @(\uparrow A,i) \le t2\ ]$$

3. For any sequence of queue operations, A = A1 ... An, the following axiom relates the start and end times of the composition to the start and end times of the individual operations:

$$\forall i\ [@(\uparrow A, i) = @(\uparrow A1, i) \wedge @(\downarrow A, i) = @(\downarrow An, i)]$$

4. For any parallel queue operations, A = A1 || ...|| An, the following axiom relates the start and end times of the composition to the start and end times of the individual operations:

$$\forall i\ [@(\uparrow A, i) = \min(@(\uparrow A1, i), ..., @(\uparrow An, i)) \wedge @(\downarrow A, i) = \max(@(\downarrow A1, i), ..., @(\downarrow An, i))]$$

5. Cycles in a repeating task do not overlap. The following two axioms express that an input operation cannot finish after the last output operation finishes, and that an output operation cannot start before the earliest input operation starts:

$$\forall i\ [\ \max(@(\downarrow out_1,i),@(\downarrow out_2,i),...,@(\downarrow out_J,i)) > \max(@(\downarrow in_1,i),@(\downarrow in_2,i),...,@(\downarrow in_K,i))\ ]$$

$$\forall i\ [\ \min(@(\uparrow out_1,i),@(\uparrow out_2,i),...,@(\uparrow out_J,i)) > \min(@(\uparrow in_1,i),@(\uparrow in_2,i),...,@(\uparrow in_K,i))\ ]$$

where J and K are the number of output and input queues, respectively.

Table 2: Axioms About Operation Start and End Times

To simplify the exposition, we introduce a simple rewrite rule: Any timing expression of the form "**repeat** *n* => E" can be rewritten as a sequence of *n* occurrences of the unguarded expression E ("E E E ... E"). Thus, the only guards we need to consider are **before**, **after**, **during**, and **when**. Table 2 gives the axioms that describe the start and end times of operations and composition of operations.

Timing Expression E =	M_{tb}***(Timing Expression)*** $M_{tb}(E)$ =
(E1)	$M_{tb}(E1)$
E1 ... En	$M_{tb}((E1\ E2)\ ...En)$
E1 ||...|| En	$\wedge\ M_{tb}(Ei \parallel Ej)$ for all $i \neq j$
E1 E2	$M_{tb}(E1) \wedge M_{tb}(E2) \wedge$ $\forall i\ [\ @(\downarrow M_{to}(E1), i) \leq @(\uparrow M_{to}(E2), i)\ ]$
E1 || E2	$M_{tb}(E1) \wedge M_{tb}(E2) \wedge$ $\forall i\ [@(\uparrow M_{to}(E1), i) < @(\downarrow M_{to}(E2), i) \wedge$ $@(\uparrow M_{to}(E2), i) < @(\downarrow M_{to}(E2), i)]$
when W => E1	$M_{tb}(E1) \wedge \forall i\ [\ W(@(\uparrow M_{to}(E1), i))\ ]$
before a1 => E1	$M_{tb}(E1) \wedge \forall i\ [\ @(\uparrow M_{to}(E1), i) \leq a1\ ]$
after a1 => E1	$M_{tb}(E1) \wedge \forall i\ [\ @(\uparrow M_{to}(E1), i) \geq a1\ ]$
during [a1, a2] => E1	$M_{tb}(E1) \wedge \forall i\ [\ a1 \leq @(\uparrow M_{to}(E1), i) \leq a2\ ]$
during [a1, r2] => E1	$M_{tb}(E1) \wedge \forall i\ [\ a1 \leq @(\uparrow M_{to}(E1), i) \leq a1 + r2\ ]$
A[r1, r2]	$\forall i\ [\ @(\uparrow A, i) + r1 \leq @(\downarrow A, i) \leq @(\uparrow A, i) + r2\ ]$
A[*, r1]	$\forall i\ [\ @(\downarrow A, i) \leq @(\uparrow A, i) + r1\ ]$
A[r1, *]	$\forall i\ [\ @(\uparrow A, i) + r1 \leq @(\downarrow A, i)\ ]$
A	true

a. M_{tb} -- Mapping from Timing Expressions to Booleans

Timing Expression E =	M_{to}***(Timing Expression)*** $M_{to}(E)$ =
loop E1	$M_{to}(E1)$
E1 ... En	$M_{to}(E1)$ ... $M_{to}(En)$
E1 ||... || En	$M_{to}(E1) \parallel ... \parallel M_{to}(En)$
guard => E1	$M_{to}(E1)$ for all guards, **when**, **before**, **during**, and **after**
A [t1, t2]	A
A	A

b. M_{to} -- Mapping From Timing Expressions to Operations

Table 3: Assigning Meaning to Timing Specifications

We assign a meaning to timing expressions by introducing a function, M_{tb} (Table 3.a), which maps timing expressions to Boolean values,

$$M_{tb} : \text{Timing Expression} \rightarrow \text{Boolean}.$$

In the definition of M_{tb} we use an auxiliary function, M_{to} (Table 3.b), which maps timing expressions to operations,

$$M_{to}: \text{Timing Expression} \rightarrow \text{Operation}.$$

M_{to} is needed because "start time" and "end time" are meaningful only for queue operations.

As an example of how to interpret the formalism intuitively, consider the entries for the **during** guard in Table 3.a. This guard specifies a time window during which the operation is allowed to start. The first time value of the window is the earliest start time allowed and must be an absolute time value. The second time value is the latest start time allowed and can be an absolute time value or a time value relative to the former. The meaning of the guarded expression is the conjunction of the meaning of the expression proper and a predicate stating the restriction on starting times.

5.2 Assigning Meaning to the Combined Specifications

Given a task description of the form:

task *taskname*
. . . . .
behavior
  requires Req ;
  ensures Ens ;
  timing E ;
. . . . .
end *taskname* ;

we give meaning to the predicates of the functional specification as related to time (i.e., at what times are these predicates to hold?) via a function M_f which maps from behavioral specifications to Boolean values:

$$M_f : \text{Predicate} \times \text{Timing Expression} \rightarrow \text{Boolean}$$

Pred.	***Expr.***	M_f***(Predicate, Timing Expression)***
Req	E	$M_f(Req,E) = \forall\, i\ [Req(@(\uparrow M_{to}(E),i)) \wedge M_{tb}(E)]$
Ens	E	$M_f(Ens,E) = \forall\, i\ [Ens(@(\downarrow M_{to}(E),i)) \wedge M_{tb}(E) \wedge Consistent(Ens,E)]$

where Consistent(Ens, E) checks to see if the **ensures** Ens predicate is meaningful with respect to the timing expression E. Consistent is defined by using two auxiliary predicates, Uses and Depends:

For all input queues q_{in}, output queues q_{out}, elements in the output queues x:

Uses: element × input queue × output queue × Predicate → Boolean

$Uses(x, q_{in}, q_{out}, Ens) =$ true, if q_{in} *appearsIn* x ∧ Ens => isIn(q_{out}, x);
false, otherwise.

$UsesSet(x, q_{out}, Ens) =$ $\{q_{in} \mid Uses(x, q_{in}, q_{out}, Ens)\}$ for all x such that isIn(q_{out}, x)

where "a *appearsIn* b" is a syntactic relation that checks if the text a occurs in the text b. Intuitively, Uses checks to see if the computation of x, the element enqueued on q_{out}, can be proven from the Ens to use any of the elements from q_{in}. In general, the element x is written in terms of a trait expression involving queue operators (e.g., first) as well as other type-specific operators (e.g,. *) as in the multiply example where x is taken to be first(in1) * first(in2).

For all input queues q_{in}, output queues q_{out}, elements in the output queues x, and for all $1 \leq i \leq$ length(q_{out}) where $i^{th}(q_{out})$ is first(rest^{i-1}(q_{out})):

Depends: element × input queue × output queue × Timing Expression → Boolean

Depends($i^{th}(q_{out})$, q_{in}, q_{out}, E) =
true, if E = E1 q_{out} E2 or E = E1 q_{out} || E2 or E = E1 || q_{out} E2, and
q_{in} *appearsIn* E1 and q_{out} *appearsIn* E1 i-1 times;
false, otherwise.

DependsSet(x, q_{out}, E) = {q_{in} | Depends(x, q_{in}, q_{out}, E) } for all x such that isIn(q_{out}, x)

Intuitively, Depends says that output elements can depend on only elements that were previously, or concurrently input.

We now define Consistent: Predicate × Timing Expression → Boolean as follows:

Consistent(Ens,E) =∀ x, ∀ q_{out} [isIn(q_{out},x) => (Uses(x,q_{out},Ens) ⊆ Depends(x,q_{out},E))]

Intuitively, we check to see that each element x in each output queue depends on only elements that have been dequeued from input queues strictly before or concurrently with the enqueueing of x.

5.3 Examples

In the absence of a timing expression, we can perform standard first-order reasoning on a functional specification. For example, if the multiply task's **ensures** predicate had the additional conjunct, first($out1_{post}$) = first(in1), then by equational reasoning (substitution of equals by equals), we see that the **ensures** predicate is satisfiable only if first(in1) * first(in2) = first(in1).

In the absence of a functional specification, we can use the axioms and rules of RTL plus our extensions listed in Section 5.1 to determine inconsistent timing expressions. For example, if the expression is in1 out1 in2, we can apply axiom 5 of Section 5.1 to show that, for each task cycle, the end of the last input operation (in2) cannot follow the end of the last output operation (out1), thus invalidating the timing expression.

```
task merge
    ports
        in1, in2: in item;
        out1: out item;
    behavior
        ensures out1post = insert(insert(out1,first(in1)),first(in2));
        timing loop (in2 out1 in1 out1);
end merge;
```

Figure 5: Merge Task

More interestingly, however, is to show how a combined specification can be proven inconsistent, where in fact, each separately is consistent and meaningful. For example, consider a task that merges data coming from two input into one output queue, as shown in Figure 5.

The **ensures** clause specifies that the output queue's items be ordered such that the item from in1 is before that from in2, but the timing expression specifies that if the item from in1 is output on the queue out1, it must be the second, not first, item in the queue (here we assume that the output queue is initially empty.) This inconsistency can be formally proven:

UsesSet(first(out1), out1, Ens) = {in1}
DependsSet(first(out1), out1, E) = {in2}

Since the UsesSet is not a subset of the DependsSet for first(out1), Consistent(Ens, E) is false.

```
task divide
    ports
        a, b: in real;
        q, r: out real;
    behavior
        ensures first(q_post) * first(a) + first(r_post) = first(b)
        timing loop (a q b r);
end merge;
```

Figure 6: Divide Task

The example in Figure 6 illustrates why subsetting and not equality is used in the definition of Consistent. It also shows the use of the Ensures predicate and the need for equational reasoning about elements in a queue (see the second conjunct in the Uses predicate).

The **ensures** clause in the Divide task specifies that the quotient of b divided by a is in q and the remainder in r; however, the timing expression says that the computation of the quotient need depend on only what is in a, and not what is in b. This inconsistency can be formally proven since:

UsesSet(first(q), q, Ens) = {a, b}
DependsSet(first(q), q, E) = {a}

More specifically, to show UsesSet(first(q), q, Ens) = {a, b} we first note that:

Uses(*quotient*(first(a), first(b)), a, q, Ens) = true
Uses(*quotient*(first(a), first(b)), b, q, Ens) = true

since a and b both "appear in" the first argument (assume *quotient* is a trait operator for real numbers.)

Using equational reasoning on the Ens, we can show

first(q) = *quotient*(first(a), first(b))

By substitution, we get

Uses(first(q), a, q, Ens) $\wedge$ Uses(first(q), b, q, Ens)

yielding:

UsesSet(first(q), q, Ens) = {a, b}

6 Examples

Figure 7 shows our multiply task with functional and timing information together. The figure shows two different multiply tasks, specified to have the same functionality but with different timing behavior. The **timing** expression in Figure 7.a states that the multiply task first checks that the input queues are non-empty, and if so perform two parallel Dequeue operations followed by an Enqueue operation. The **timing** expression in Figure 7.b states that the inputs come in sequentially instead of in parallel.

```
task multiply
  ports
    in1, in2: in matrix;
    out1: out matrix;
  behavior
      requires  number_of_rows(first(in1)) = number_of_cols(first(in2));
      ensures   out1post = insert(out1, first(in1) * first(in2));
      timing when (~isEmpty(in1) and ~isEmpty(in2)) =>
                  ((in1.Dequeue || in2.Dequeue) delay[10,15] out1.Enqueue);
end multiply;
```

a. Parallel Input

```
task multiply
  ports
    in1, in2: in matrix;
    out1: out matrix;
  behavior
      requires  number_of_rows(first(in1)) = number_of_cols(first(in2));
      ensures   out1post = insert(out1, first(in1) * first(in2));
      timing when (~isEmpty(in1) and ~isEmpty(in2)) =>
                  (in1.Dequeue in2.Dequeue delay[10,15] out1.Enqueue);
end multiply;
```

b. Serial Input

Figure 7: Matrix Multiplication Task

To further illustrate the richness of our specification language and to show the benefits of cleanly separating the functional from the timing information, we write three alternative descriptions for a task built into our library. This task, **deal**, has one input port and a number of output ports. Data dequeued from the input port is enqueued to one of the output ports, but this can be implemented in a number of ways, as illustrated in Figure 8.

In the examples, we will drop the name of the queue operation and use just the name of the port (i.e., in1 instead of in1.Dequeue). Since this paper introduces only two queue operations: Enqueue and Dequeue, and given that the former applies only to input queues and the other applies only to output queues, no confusion should occur as to which operation is implied.

The first example (Figure 8.a) states that we alternate the dequeuing of input and enqueueing of output and ensures that the first (second) output queue will see the first (second) element removed from the input queue. The second example (Figure 8.b) states that we dequeue all input before the output operations start, which themselves take place concurrently. It allows for the first dequeued element to be enqueued on either of the output queues, but ensures that the second dequeued element will not be enqueued to the same as the first. The third example (Figure 8.c) states that input data are dequeued and grouped in pairs before enqueueing them into the output ports. The first pair is enqueued to the first output queue; the second pair, to the second.

```
task deal
    ports
        in1: in matrix;
        out1, out2: out matrix;
    behavior
        ensures
            out1post = insert(out1, first(in1)) &
            out2post = insert(out2, second(in1));
        timing loop (in1 out1 in1 out2);
end deal;
```

a. Alternating Input and Output

```
task deal
    ports
        in1: in matrix;
        out1, out2: out matrix;
    behavior
        ensures
            (out1post = insert(out1,first(in1)) &
             out2post = insert(out2, second(in1))) |
            (out2post = insert(out2, first(in1)) &
             out1post = insert(out1, second(in1)))
        timing loop (in1 in1 (out1 || out2))
end deal;
```

b. Concurrent Output

```
task deal
    ports
        in1: in matrix;
        out1, out2: out matrix;
    behavior
        ensures
            out1post = insert(insert(out1, first(in1)), second(in1)) &
            out2post = insert(insert(out2, third(in1)), fourth(in1))
        timing loop (in1 in1 in1 in1 (out1 || out2) (out1 || out2))
end deal
```

c. Grouping Data

Assume that second(in1), third(in1), and fourth(in1) as abbreviations for first(rest(in1)), first(rest(rest(in1))), first(rest(rest(rest(in1)))), respectively, are defined in the trait for queues.

Figure 8: Deal Task

7 Related Work

The axiomatic approach to specifying a program's functional behavior has its origins in Hoare's early work on verification [17] and later work on proofs of correctness of implementations of abstract data types [18], where first-order predicate logic pre- and post-conditions are used for the specification of each operation of the type. The algebraic approach, which defines data types to be heterogeneous algebras [5], uses axioms to specify properties of programs and abstract data types, but the axioms are restricted to equations. Much work has been done on algebraic specifications for abstract data types [12, 11, 14, 6]; we use more recent work on Larch specifications [16] for program modules. None of this work addresses the formal specification of timing behavior of systems.

Operational approaches, such as those based on Timed Petri-net models [23, 26], are more commonly used for specifying behavior of real-time systems. Timed Petri-nets can be roughly characterized by whether "operation" time is assigned to the *transitions*, as in the original model by Ramchandani [23], or is assigned to the *places*, as in Sifakis' model [26]. In addition, both deterministic and stochastic timing are allowed, giving origin to a variety of models for specifying or evaluating performance requirements. This has been illustrated in recent work by Coolahan [9] (places, deterministic), Smith [27] (transitions, deterministic), Wong [28] (places, stochastic), and Zuberek [29] (transitions, stochastic). In contrast, our work takes a more axiomatic than operational approach to specifying timing behavior.

Specification and verification of timing requirements for real-time systems include recent work by Dasarthy [10], and by Lee, Gehlot, and Zwarico [20, 30]. This work as well as that by Jahanian and Mok, whose real-time logic we borrow, all focus on timing properties and not on functional behavior. Either states are left uninterpreted or predicates on states are simplistic, e.g., Boolean modes as in Jahanian and Mok's work. In contrast, since we have a formal means of specifying the functional behavior of tasks and the data on which they operate, we have a more expressive specification language with a richer semantics.

The programming model we have in mind for the developers of real time, concurrent applications is based on data flowing between computing elements. However, we do not impose a data driven computation model, a basic premise of most data flow languages [1]. Tasks in our applications are asynchronous and operate on their input and output queues according to a regime described by each task's timing expression. These requirements are difficult to satisfy in traditional data flow languages although a recent data flow language, LUSTRE, overcomes these limitations. LUSTRE [4, 8] supports the concept of timing in a stream language. LUSTRE is based on LUCID [2] with the addition of timing operators and user defined clocks associated with the variables (sequences of values). In the original version of the language [4] functions and operators required that all input variables be associated with the same clock (i.e., the input and output streams moved in lock-step). These restrictions have been relaxed in the latest version of the language [8]

8 Summary

Our approach to specifying the functional and timing behavior of real-time applications for a heterogeneous machine has the following characteristics:

- It takes advantage of two well defined, though isolated, pieces of previous work.
- There is a clean separation of concerns between the two specifications.
- The semantics of both specifications as well as their combination are simple.

In our language design, we strove to separate the functional specification from the timing specification so that a task's functionality could be understood independently of its timing behavior. This separation of concerns gives us the usual advantages of modularity. Different timing specifications can be attached to the same functional specification. Task implementors can focus on satisfying functionality first, timing second. Task validation can be performed separately. For example, one could use formal verification for functionality and simulation for timing. However, we are not completely satisfied with our definition of Consistent(Ens, E) (Section 5.2) since it depends on a syntactic relation, appearsIn, which is easy to check for but is probably too restrictive.

Since the semantics can be given in terms of first-order predicate logic, our specifications are amenable to machine manipulation and analysis. The algebraic style of Larch traits can be analyzed by rewrite-rule tools, e.g., Reve [21]; the two-state predicates of Larch interfaces and thus, task predicates, can be analyzed by verification systems that support first-order reasoning, e.g., Gypsy, HDM, and FDM [13, 24, 25]; formulae in real-time logic can be mechanically transformed into equivalent formulae in Presburger arithmetic [19]. However, though many of these tools are available, much work is needed to integrate them so our specifications could be fully machine-checked and analyzed.

Acknowledgements

We thank Al Mok for his assurance that we are using RTL properly and the anonymous referee for bringing to our attention the most recent development of LUSTRE.

References

[1] W.B. Ackerman.
Data Flow Languages.
IEEE-CS Computer 15(2), February, 1982.

[2] E.A. Ashcroft and W.W. Wadge.
LUCID: The Data Flow Programming Language.
Academic Press, 1985.

[3] M.R. Barbacci and J.M. Wing.
Durra: A Task-level Description Language.
Technical Report CMU/SEI-86-TR-3, Software Engineering Institute, Carnegie Mellon University, 1986.

[4] J-L. Bergerand, P. Caspi, N. Halbwachs, D. Pilaud, and E. Pilaud.
Outline of a Real Time Data Flow Language.
In *Procedings of the IEEE-CS Real Time Systems Symposium*, pages 33-42. IEEE Computer Society Press, December, 1985.

[5] G. Birkhoff and J.D. Lipson.
Heterogeneous Algebras.
Journal of Combinatorial Theory 8:115-133, 1970.

[6] R.M. Burstall, and J.A. Goguen.
Putting Theories Together to Make Specifications.
In *Fifth International Joint Conference on Artificial Intelligence*, pages 1045-1058. August, 1977.
Invited paper.

[7] R.H. Campbell and A.N. Habermann.
The Specification of Process Synchronization by Path Expressions.
Lecture Notes in Computer Science.
Springer-Verlag, 1974, pages 89-102.

[8] P. Caspi, D. Pilaud, N. Halbwachs, and J.A. Plaice.
LUSTRE: A Declarative Language for Programming Synchronous Systems.
In *Procedings of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 178-188. ACM, January, 1987.

[9] J.E. Coolahan and N. Roussopulos.
A Timed Petri Net Methodology for Specifying Real-Time System Requirements.
In *International Workshop on Timed Petri Nets*, pages 24-31. IEEE Computer Society Press, Torino, Italy, July, 1985.

[10] Dasarthy.
Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them.
IEEE Transactions on Software Engineering 11(1):80-86, January, 1985.

[11] H. Ehrig and B. Mahr.
Fundamentals of Algebraic Specification 1.
Springer-Verlag, 1985.

[12] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright.
Abstract Data Types as Initial Algebras and Correctness of Data Representations.
In *Proceedings from the Conference of Computer Graphics, Pattern Recognition and Data Structures*, pages 89-93. ACM, May, 1975.

[13] D.I. Good, R.M. Cohen, C.G. Hoch, L.W. Hunter, and D.F. Hare.
Report on the Language Gypsy, Version 2.0.
Technical Report ICSCA-CMP-10, Certifiable Minicomputer Project, The University of Texas at Austin, September, 1978.

[14] J.V. Guttag.
The Specification and Application to Programming of Abstract Data Types.
PhD thesis, University of Toronto, Toronto, Canada, September, 1975.

[15] J.V. Guttag, J.J. Horning, and J.M. Wing.
Larch in Five Easy Pieces.
Technical Report 5, DEC Systems Research Center, July, 1985.

[16] J.V. Guttag, J.J. Horning, and J.M. Wing.
The Larch Family of Specification Languages.
IEEE Software 2(5):24-36, September, 1985.

[17] C.A.R. Hoare.
An axiomatic basis for computer programming.
Communications of the ACM 12(10):576-583, October, 1969.

[18] C.A.R. Hoare.
Proof of Correctness of Data Representations.
Acta Informatica 1(1):271-281, 1972.

[19] F. Jahanian and A.K. Mok.
Safety Analysis of Timing Properties in Real-Time Systems.
IEEE Transactions on Software Engineering 12(9):890-904, September, 1986.

[20] I. Lee, and V. Gehlot.
Language Constructs for Distributed Real-Time Programming.
In *Proceedings of the Real-Time Systems Symposium*, pages 57-66. San Diego, December, 1985.

[21] P. Lescanne.
Computer Experiments with the REVE Term Rewriting System Generator.
In *Proceedings of Tenth Symposium on Principles of Programming Languages*, pages 99-108. ACM, Austin, Texas, January, 1983.

[22] D.R. Musser.
Abstract Data Type Specification in the Affirm System.
IEEE Transactions on Software Engineering 6(1):24-32, January, 1980.

[23] C. Ramchandani.
Analysis of Asynchronous Concurrent Systems by Petri Nets.
Technical Report TR-120, MIT Project MAC, 1974.

[24] L. Robinson, and O. Roubine.
SPECIAL - A Specification and Assertion Language.
Technical Report CSL-46, Stanford Research Institute, Menlo Park, Ca., January, 1977.

[25] J. Scheid and S. Anderson.
The Ina Jo Specification Language Reference Manual.
Technical Report TM-(L)-6021/001/00, System Development Corporation, Santa Monica, CA, March, 1985.

[26] J. Sifakis.
Use of Petri Nets for Performance Evaluation.
In *Proceedings of the IFIP Third International Workshop on Modeling and Performance Evaluation of Computer Systems*, pages 75-93. North-Holland Publishing Co., Amsterdam, The Netherlands, 1977.

[27] C.U. Smith.
Robust Models for the Performance Evaluation of Hardware/Software Designs.
In *International Workshop on Timed Petri Nets*, pages 172-180. IEEE Computer Society Press, Torino, Italy, July, 1985.

[28] C.Y.Wong, T.S. Dillon, and K.E. Forward.
Timed Places Petri Nets With Stochastic Representation of Place Time.
In *International Workshop on Timed Petri Nets*, pages 96-103. IEEE Computer Society Press, Torino, Italy, July, 1985.

[29] W.M. Zuberek.
Performance Evaluation Using Extended Timed Petri-nets.
In *International Workshop on Timed Petri Nets*, pages 272-278. IEEE Computer Society Press, Torino, Italy, July, 1985.

[30] A. Zwarico and I. Lee.
Proving a Network of Real-time Processes Correct.
In *Proceedings of Real-Time Systems Symposium*, pages 169-177. San Diego, December, 1985.