

# Classification Schemes to Aid in the Analysis of Real-Time Systems

Paul Z. Kolano  
Plenar Corporation  
Silicon Valley Operation  
303 Almaden Blvd., Suite 600  
San Jose, CA 95110  
+1 408 938 5770  
paul.kolano@plenar.com

Richard A. Kemmerer  
Reliable Software Group  
Computer Science Department  
University of California  
Santa Barbara, CA 93106  
+1 805 893 4232  
kemm@cs.ucsb.edu

## ABSTRACT

This paper presents three sets of classification schemes for processes, properties, and transitions that can be used to assist in the analysis of real-time systems. These classification schemes are discussed in the context of ASTRAL, which is a formal specification language for real-time systems. Eight testbed systems were specified in ASTRAL, and their proofs were performed to determine proof patterns that occur most often. The specifications were then examined in an attempt to derive specific characteristics that could be used to statically identify each pattern within a specification. Once the classifications were obtained, they were then used to provide systematic guidance for analyzing real-time systems by directing the prover to the proof techniques most applicable to each proof pattern. This paper presents the set of classification schemes that were developed and discusses how they can be used to assist the proof process.

## Keywords

Formal methods, formal specification and verification, real-time systems, ASTRAL, system classification, analysis guidance, timing requirements.

## 1. INTRODUCTION

Real-time systems are distributed, concurrent, and reactive, which makes the proofs of their critical requirements extremely complex. These proofs are most often discharged based solely on the ingenuity of the human prover. To make the analysis of real-time systems less ad hoc and less dependent on human ingenuity, it is desirable to provide a methodology that can be used to direct the proofs in a systematic fashion. To develop such a methodology, it is necessary to define a set of classification schemes that can be used to differentiate between various specification and proof

styles. These classifications are used to separate specifications with different patterns as well as separating individual proofs into simpler pieces. It is critical not only that different classifications result in different proof styles, but also that the classifications are statically recognizable so that they can be consulted before analysis begins. When a user analyzes a real-time system, system entities can be classified appropriately and then the guidelines for each applicable classification can be consulted to direct the proofs. These guidelines include which analysis step should be performed next, how it can be performed efficiently using the appropriate tools and techniques, and how the results of different approaches can be used to complement each other.

In order to determine the classification schemes that are most useful during analysis, a set of testbed systems was developed. These systems consist of a variety of different process and property types. Each system was specified in ASTRAL [2], which is a formal specification language for real-time systems. The proofs of each system were performed to determine the proof patterns that occurred most often, which could then be reused in the proofs of other systems based on the classification schemes. The specifications that comprised the testbed varied from the specification of a distributed mutual exclusion protocol to a phone switching system to a production facility. More specifically, the specifications include a number of standard benchmark systems: a bakery specification that describes the distributed mutual exclusion algorithm of [13], a cruise control system based on the description in [20], an elevator control system adapted from [6], a production cell specification based on the description in [14], a railroad crossing system based on the description in [7], and a stoplight specification adapted from the stoplight control system described in [6]. The testbed also includes the specification of an electronic scoring system for Olympic boxing based on a description of the system taken from the official 1996 Olympic web site [17]. Finally, the testbed includes a long distance telephony specification taken from [2].

These specifications represent a wide variety of real-time systems. There are small systems such as the railroad crossing and large systems such as the stoplight control system. There are simple systems such as the cruise control and complex systems such as the phone system. There are open systems such as the scoring system and closed systems such as the bakery algorithm. There

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISSTA'00, Portland, Oregon.

Copyright 2000 ACM 1-58113-266-2/00/0008...\$5.00.

are deterministic systems such as the production cell and nondeterministic systems such as the elevator control system. Finally, there are systems where assumptions are not needed to complete the proofs such as the stoplight control system and systems where assumptions are needed such as the phone system. Given that the testbed systems represent such a wide variety of real-time systems, it is expected that the classifications developed for these systems are applicable to most systems in general. In addition to their usefulness during the development of the systematic analysis methodology, as a side result, these systems demonstrate the flexibility and expressiveness of ASTRAL. The complete specifications of the testbed systems can be found in [9].

This paper presents the classification schemes that were developed based on the set of testbed systems. Although these classification schemes and proof techniques were developed based on ASTRAL specifications, it is felt that the results for ASTRAL are applicable to many other real-time specification languages by just taking into account the differences in terminology and syntax. Classification schemes were developed for processes, properties, and transitions. Each scheme is presented as well as how each classification is of assistance during the proof process. Due to space limitations, the proof techniques are discussed at a high level. For complete technical details on the systematic analysis methodology, see [11]. The next section gives a brief overview of the ASTRAL language. Sections 3, 4, and 5 discuss the classification schemes for processes, properties, and transitions, respectively, and how each assists analysis. Section 6 presents some related work in classification and systematic guidance. Finally, some conclusions and future directions for this research are discussed in section 7.

## 2. ASTRAL OVERVIEW

In ASTRAL, a real-time system is described as a collection of state machine specifications, where each specification represents a process type of which there may be multiple, statically generated, instances. Each process instance in the system executes concurrently and asynchronously with all the other process instances. Additionally, a *global specification* contains declarations for types and constants that are shared among more than one process type, as well as assumptions about the global environment and critical requirements for the whole system.

An ASTRAL *process specification* consists of a sequence of *levels*. Each level is an abstract data type view of the process being specified. The first (“top level”) view is a very abstract model of what constitutes the process (types, constants, variables), what the process does (state transitions), and the critical requirements the process must meet (invariants and schedules). Lower levels are increasingly more detailed with the lowest level corresponding closely to high-level code. Figure 1 shows one of the process types of the Olympic boxing scoring system. The Judge process represents the scoring panel that judges use to keep score of the boxing match.

The process being specified is thought of as being in various *states*, with one state differentiated from another by the values of its *state variables*, which can be changed only by means of *state transitions*. Every process can export both state variables and transitions; as a consequence, the former are readable by other processes while the latter are executable from the external

environment. Processes communicate by broadcasting the values of exported variables and the start and end times of exported transitions. In the Judge process, the Score\_Card variable and the Score transition are exported. Judge imports the global constant Window, the process ID Time\_Keeper, and the global types Pos\_Real, Non\_Negative, and Boxer from the global specification. In addition, the In\_Round variable is imported from the Time\_Keeper instance of the Timer process type.

Transitions are described in terms of entry and exit assertions by using an extension of first-order predicate calculus. Transition *entry assertions* describe the constraints that state variables must satisfy in order for the transition to fire, while *exit assertions* describe the constraints that are fulfilled by state variables after the transition has fired. An explicit non-null duration is associated with each transition.

Each transition is either a local transition or an exported transition. A local transition is enabled when its entry assertion is satisfied. An exported transition, however, is only enabled when both its entry assertion is satisfied and when it has been called (i.e. invoked) from the external environment. Transitions are executed as soon as they are enabled assuming no other transition for that process instance is executing. If two or more transitions are enabled simultaneously, a nondeterministic choice will occur and only one of them will execute. In the Judge process, the Score transition is enabled when it has been called from the external environment but not yet serviced, when the match is in a round, and when the last time Score fired was more than Window time units in the past.

In addition to specifying system state (through process variables and constants) and system evolution (through transitions), an ASTRAL specification also defines system critical requirements and assumptions about the behavior of the environment that interacts with the system. The behavior of the environment is expressed by means of *environment clauses*, which describe assumptions about the pattern of invocation of exported transitions. Critical requirements are expressed by means of *invariants* and *schedules*. Invariants represent requirements that must hold in every state reachable from the initial state, no matter what the behavior of the external environment is, while schedules represent additional properties that must be satisfied provided that the external environment behaves as assumed.

Invariants and schedules are proved over all possible executions of a system. A system execution is a set of process executions that contains one process execution for each process instance in the system. A process execution for a given process instance is a history of events on that instance. The value of an expression E at a time t1 in the history can be obtained using the *past* operator, *past(E, t1)*, such that  $t1 \leq \text{now}$ , where *now* is a special global variable used to denote the current time in the system. There are four types of events that may occur in an ASTRAL history. A *call event*, *Call(tr1, t1)*, occurs for an exported transition tr1 at a time t1 iff tr1 was called from the external environment at t1. A *start event*, *Start(tr1, t1)*, occurs for a transition tr1 at a time t1 iff tr1 fires at t1. Similarly, an *end event*, *End(tr1, t1)*, occurs iff tr1 ends at t1. Finally, a *change event*, *Change(v1, t1)*, occurs for a variable v1 at a time t1 iff v1 changes value at t1. Note that change events can only occur when an end event occurs for some transition.

<pre> PROCESS SPECIFICATION Judge IMPORT   Pos_Real, Boxer, Time_Keeper.In_Round,   Non_Negative, Window, Time_Keeper EXPORT   Score, Score_Card CONSTANT   Score_Dur: Pos_Real VARIABLE   Score_Card(Boxer): Non_Negative AXIOM   Score_Dur &lt; Window INITIAL   FORALL B: Boxer     (Score_Card(B) = 0) SCHEDULE   Call(Score, now)   → Start(Score, now) </pre>	<pre> ENVIRONMENT ( EXISTS t: Time   ( t ≤ Now     &amp; Call<sub>2</sub>(Score, t)   → Call(Score) - Call<sub>2</sub>(Score) ≥ 2 * Window)   &amp; ( Call(Score, now)     → Time_Keeper.In_Round) TRANSITION Score(B: Boxer) ENTRY [TIME: Score_Dur]   Time_Keeper.In_Round   &amp; FORALL t: Time     ( t &lt; Now       &amp; past(Start(Score, t) , t)     → Now - t &gt; Window) EXIT   Score_Card(B) BECOMES   Score_Card'(B) + 1 END Judge </pre>
---	--

Figure 1. The Judge process

Using these operators, a variety of complex properties can be expressed. For example, the schedule of the Judge process states that whenever Score is called from the external environment (i.e. a human judge presses the score button for a particular boxer on the scoring panel), Score will start immediately. This property can only be guaranteed when the environment holds, which states that consecutive calls to Score will be separated by  $2 * \text{Window}$  time units and that calls to Score will only occur when the match is in a round. An introduction and complete overview of the ASTRAL language can be found in [2].

ASTRAL is supported by the ASTRAL Software Development Environment (SDE) [12], which allows the user to edit and manage complex specifications as well as providing support for their analysis. In particular, proof obligations can be automatically produced from specifications, and proofs are supported by both model checking and deductive facilities. The model checker can check the critical requirements of a particular instance of a specification over a finite time interval. ASTRAL has been encoded into the language of the PVS theorem prover [3] to support deductive reasoning.

### 3. PROCESS CLASSIFICATION

The goal of process classification is to identify detectable differences in process behavior that lead to significantly different styles of proofs. By examining the proofs of the testbed systems, three such process classifications were identified that can be statically recognized. These classifications are multi-threaded processes, iterative single-threaded processes and simple single-threaded processes. Each of these classifications is discussed in the following subsections.

### 3.1 Multi-Threaded Processes

A multi-threaded process is a process in which multiple threads of execution are occurring at any given time in the process. For example, consider the central control process of the phone system. In this case, the “threads” are the servicing of each phone in the central control’s area. Each thread consists of a chain of actions that occurs during the evolution of a call for a particular phone. The central control can only perform one stage of a single phone’s thread at any given time. The phone threads of all the phones are interleaved with each other in the central control. Figure 2 shows the threads of two phones and how they might be interleaved on the central control. The labels GDT, PD, PC, etc. are abbreviations for the central control transitions Give\_Dial\_Tone, Process\_Digit, Process\_Call, Enable\_Ringback, Disable\_Ringback\_Pulse, Start\_Talk, and Terminate\_Call.

The key fact about multi-threaded processes is that there are multiple independent threads interleaved on the process, thus at any given time, any combination of thread stages may be enabled in the set of threads. This means that there is essentially no way to guarantee a real-time response property of a single thread unless some set of restrictions is placed on the behavior of the complete set of threads. Namely, it is necessary to guarantee that the thread will not be “starved” by the other threads. This is usually achieved by choosing an appropriate *scheduling policy* (e.g. fixed priority, FIFO, round robin, etc.) and by placing various limitations on the number of transitions enabled or threads that require service at any given time. In multi-threaded operating systems, the scheduling policy specifies how to select the next thread to execute on the processor when multiple threads are waiting for service. In ASTRAL, a scheduling policy can be implicitly specified by the *transition selection clause*, which states how to select the next transition to execute on a process when multiple transitions are enabled.

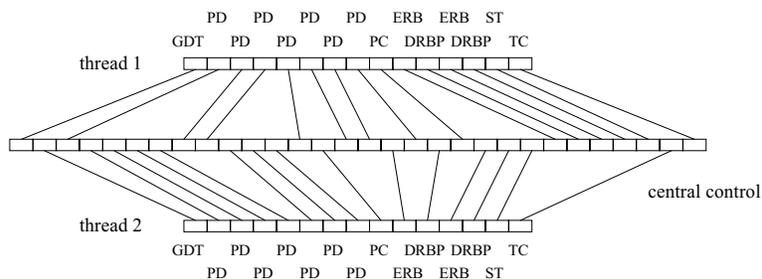


Figure 2. Interleaved phone threads on the central control

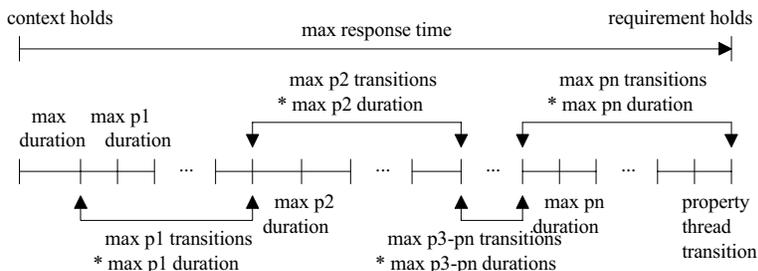


Figure 3. Deriving the maximum response time for fixed priority scheduling

After the scheduling policy has been determined and the appropriate limits have been found, an estimate of the maximum response time can be derived based on the scheduling policy of the process. For example, in fixed priority scheduling, the maximum response time can be derived as shown in figure 3. In this case, there may be some arbitrary transition firing when all the requests are made. Thus, before the fixed priority scheduling policy takes effect, there may be a delay of up to the duration of the slowest transition. In the figure, there are  $n - 1$  priority levels that are higher than the priority level of the thread associated with the requirement of the property. Thus, before the thread of interest can fire, a bounded number of threads at each of the  $n - 1$  higher priority levels can potentially fire.

### 3.2 Iterative Single-Threaded Processes

An iterative single-threaded process is a process that repeatedly executes a sequence of actions in a countable fashion. That is, a similar sequence of actions is performed during each iteration. Note that almost all ASTRAL processes are cyclic in some way. In an iterative process, however, there is some record of how many iterations have been performed that affects the behavior of the process. The count may represent floors in a building, loop counts in a program, etc.

For example, consider the Elevator process of the elevator control system. The Elevator process iterates over the position of the elevator car in the building. At each floor, the elevator performs a specific sequence of actions depending on the position, the direction of movement, and the requests outstanding in the building. For example, if the elevator just arrived at a floor  $i$  moving up and there are requests on floors  $i$  and  $i+1$ , the sequence of actions would be door\_open, door\_stop, door\_close, door\_stop, move\_up, and arrive.

The first thing to notice in iterative single-threaded processes is that in order for a response requirement to be guaranteed, the maximum time that can be spent in any iteration must be bounded. The other thing is that the number of full iterations between when the context holds and when the requirement is to hold must also be bounded. Thus, the proofs of response requirements can be performed by determining these bounds, deriving the maximum response time accordingly, and then checking the derived time against the required time. For a property in which the requirement must hold within a given period of time after the context holds, the maximum response time can be derived as shown in figure 4. First, it is necessary to derive how long it takes to reach a full iteration from when the context holds. Then, the maximum time of each full iteration is calculated and multiplied by the maximum number of full iterations. Finally, how long it takes to reach a time at which the requirement holds from the last full iteration is derived.

### 3.3 Simple Single-Threaded Processes

The simple single-threaded processes are the processes that are neither multi-threaded processes nor iterative single-threaded processes. These processes do not necessarily exhibit “simple” behavior. Rather, a cycle of a simple single-threaded process’s execution represents the interval over which properties in the process must be proved. This is in contrast to an iterative single-threaded process in which a property may need to be proved over multiple cycles of the process’s execution. An example of a simple single-threaded process is the press component of the production cell, which cyclically moves from a loading position to a forging position to an unloading position as shown in figure 5.

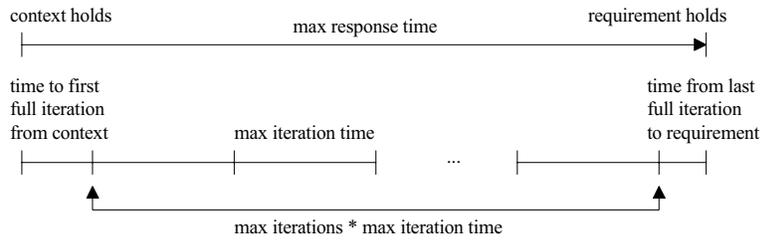


Figure 4. Deriving the maximum response time

### 3.4 Testbed Process Classifications

Table 1 shows the process classification of each process in the testbed systems, where MT is multi-threaded, IST is iterative single-threaded, and SST is simple single-threaded. The numbers were obtained using the process classification component of the ASTRAL SDE. The process classifier automatically classifies a process by examining its transitions and attempting to identify those with multi-threaded or iterative behavior, which indicates that the process as a whole is multi-threaded or iterative. A large percentage of the total number of processes were of the more straightforward simple single-threaded variety meaning that the complexities associated with the other two types can be avoided in most cases.

## 4. PROPERTY CLASSIFICATION

To describe the property classifications, it is first necessary to introduce a few definitions. Every first-order logic formula can be written in the form “context  $\rightarrow$  requirement”, where the *context* is a conjunction of unnegated conditions and the *requirement* is a disjunction of unnegated conditions. The context describes the conditions under which the requirement must hold. That is, the requirement is not required to hold under any conditions in which the context does not hold. A property may have an empty context, “TRUE  $\rightarrow$  requirement”, or an empty requirement, “context  $\rightarrow$  FALSE”.

The *context times* are the times that are referenced in some timed operator expression (i.e. past, change, start, end, or call in ASTRAL) in the conditions of the context. In languages other than ASTRAL, this definition would reflect the differences in timed operators. These times may be concrete times such as now - 5 (in ASTRAL), or symbolic times such as a quantified time variable  $t$  that has been restricted in some way. The *requirement*

*times* are the times that are referenced in a similar expression in the conditions of the requirement. For example, consider the following property of the Speed\_Control process of the cruise control system.

```

control_dur ≤ input_dur
& now ≥ input_dur + input_dur
& past(maintaining_speed, now - input_dur - input_dur)
& call(set_brake_pedal, now - input_dur - input_dur)
→ EXISTS t: Time
  ( now - input_dur - input_dur ≤ t
  & t ≤ now
  & ~past(maintaining_speed, t))

```

In this property, the only context time is now - input\_dur - input\_dur and the only requirement time is  $t$ . Properties can be classified based on the forms and the times of their context and requirement. The following sections describe these classifications, which are untimed, timed forward safety, timed backward safety, timed forward liveness, and timed backward liveness. Every ASTRAL property falls into one of these five classifications.

### 4.1 Untimed Properties

An untimed property is a property in which the only context and requirement time is the current time (i.e. now in ASTRAL). The most common form of an untimed property is one that consists solely of boolean combinations of local state variables. For example, the property shown below is an untimed property of the Controller process of the stoplight control system.

```

FORALL d: direction
  ( circle(d) = green
  → arrow(opp(d)) = red)

```

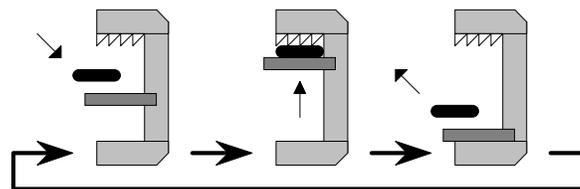


Figure 5. A cycle of the production cell press

**Table 1. Process classifications of testbed systems**

System	Process Type	MT	IST	SST
Bakery Algorithm	Proc		X	
Cruise Control	Accelerometer			X
	Speed_Control			X
	Speedometer			X
	Tire_Sensor			X
Elevator	Elevator		X	
	Elevator_Button_Panel			X
	Floor_Button_Panel			X
Olympic Boxing	Judge			X
	Tabulate		X	
	Timer		X	
Phone	Central_Control	X		
	Phone			X
Production Cell	P_Crane			X
	P_Deposit			X
	P_Deposit_Sensor			X
	P_Feed			X
	P_Feed_Sensor			X
	P_Press			X
	P_Robot			X
	P_Table			X
Railroad Crossing	Gate			X
	Sensor			X
Stoplight	Controller	X		
	Sensor			X
<b>Total</b>	<b>23</b>	<b>2</b>	<b>4</b>	<b>19</b>

In this property, whenever the circle of a direction is green, the arrow of the opposing direction must be red.

This property classification is significant because it allows the use of *frame axioms* [1], which assert that elements of the state that are not explicitly changed to a new value in the postcondition of a transition remain unchanged from the precondition. Normally in timed transition systems, the current time is an element of the state. Thus, since time is continuously changing, it is not possible to use a frame axiom because even though a property may hold when a transition started and when it ended, the property may have been violated in between due to a change in time. Untimed properties do not depend on the current time, however, thus they cannot be violated while a transition is executing, but only when a transition changes the state. This means that it is sometimes possible to prove these properties simply by examining the entry and exit assertions of each transition and assuring that (1) any time the context is set to true in an exit assertion, the requirement is set to true in the exit assertion or is true in the entry assertion and is unchanged in the exit assertion and (2) any time the requirement is set to false in an exit assertion, the context is set to false in the exit assertion or is false in the entry assertion and is unchanged in the exit assertion. For example, consider the `maintain_speed` transition of the `Speed_Control` process of the cruise control system.

```

TRANSITION maintain_speed
ENTRY [TIME: input_dur]
    cruise_on
    & ~maintaining_speed
EXIT
    cruise_throttle = throttle'
    & desired_speed = the_speedometer.speed
    & maintaining_speed

```

This transition preserves the property “`maintaining_speed` → `cruise_on`” because when the context is set to true in the exit assertion (i.e. `maintaining_speed` is asserted), the requirement is true in the entry assertion (i.e. `cruise_on` is asserted) and is unchanged in the exit assertion. This procedure has been fully automated in the ASTRAL theorem prover [10].

## 4.2 Timed Properties

A timed property is either a *safety* property or a *liveness* property and is either *forward* or *backward*. In a safety property, the requirement must hold at all times in an interval of time. In a liveness property, the requirement must hold at least once in an interval of time. If a property is forward, then there is some context time that is less than or equal to every requirement time. If a property is backward, then there is some requirement time that is less than or equal to every context time.

For example, the property shown below is a forward liveness property of the P\_Robot process of the production cell.

```
FORALL t: Time
  ( start(Arm1_Drop, now)
    & end(Arm1_Drop, t)
  → EXISTS t1: Time
    ( t < t1
      & t1 < now
      & end(Arm2_Pickup, t1)))
```

This property states that between any two consecutive drops of an object by arm one, arm two must pickup an object. This property is a forward property because there is context time,  $t$ , that is less than or equal to every requirement time. It is a liveness property because the requirement is only required to hold once in the interval  $(t, \text{now})$ , namely at  $t1$ .

As another example, the property shown below is a backward safety property of the Sensor process of the railroad crossing.

```
FORALL t: Time
  ( change(train_in_R, now)
    & ~train_in_R
    & now - ((dist_R_to_I + dist_I_to_out)
              / max_speed - response_time) ≤ t
    & t < now
  → past(train_in_R, t))
```

This property states that whenever a sensor stops reporting a train in the region, the sensor has been reporting a train for at least the past  $(\text{dist\_R\_to\_I} + \text{dist\_I\_to\_out}) / \text{max\_speed} - \text{response\_time}$  time. This property is a backward property because there is a requirement time,  $t$ , that is less than or equal to every context time. It is a safety property because the requirement is required to hold at all times in the interval  $[\text{now} - ((\text{dist\_R\_to\_I} + \text{dist\_I\_to\_out}) / \text{max\_speed} - \text{response\_time}), \text{now})$ .

These classifications correspond closely to the temporal logic operators presented in [16]. Forward safety and forward liveness correspond to the *henceforth* and *eventually* operators, while backward safety and backward liveness correspond to the *has-always-been* and *once* operators.

The distinction between proving forward properties and proving backward properties occurs due to nondeterminism. When a forward property is proved, all possible nondeterministic choices that can be made in the system, such as events occurring or not

occurring in the external environment and choices between which transition will fire, must be considered. When a backward property is proved, however, the path along which the system has evolved has already been determined; thus, nondeterminism is usually less of a factor.

The distinction between proving liveness properties and proving safety properties is that for liveness properties, the prover must show all of the exact sequences of events that occur from the time the context holds until the time the requirement holds in order to assure that the required event will occur in every given scenario. When proving a safety property, however, it can be assumed that some “bad event” has occurred and then the conditions that hold in the proof interval can be used to produce a contradiction. In general, this is usually easier than showing all possible evolutions of the system since much of the detail of the executions can be abstracted away.

### 4.3 Testbed Property Classifications

Table 2 shows the number of each type of property in each of the testbed systems, where U is untimed, F/B is forward/backward, and S/L is safety/liveness. The numbers were obtained by completely splitting all of the formulas of each specification into conjunctive normal form and then invoking the property classification component of the ASTRAL SDE on each split. The property classifier automatically classifies a formula by transforming it to a canonical form and then analyzing its quantifiers and context/requirement times appropriately. The properties are broken down into the properties that occur in the requirements sections (i.e. invariant, schedule, and constraint clauses) and those that occur in the assumptions sections (i.e. environment and imported variable clauses). The results indicate that a large portion of the properties are untimed properties meaning that simpler techniques can be applied to these properties and more concentration can be given to the more difficult properties, which are the timed properties.

## 5. TRANSITION CLASSIFICATION

While process and property classifications are high-level classification schemes that identify distinct proof styles, a transition classification is a low level scheme that facilitates steps within each proof. Namely, a transition classification can be used to determine how the next or previous transition is found in a particular execution and how much time elapses or has elapsed since that transition fired.

**Table 2. Property classifications of testbed systems**

System	Requirements					Assumptions					Total
	U	FS	FL	BS	BL	U	FS	FL	BS	BL	
Bakery Algorithm	11	1	0	0	1	3	1	0	0	1	<b>18</b>
Cruise Control	5	0	2	0	0	0	0	0	0	0	<b>7</b>
Elevator	8	0	8	0	3	2	9	0	0	3	<b>33</b>
Olympic Boxing	8	2	0	0	1	1	5	0	0	0	<b>17</b>
Phone	26	14	0	0	0	0	8	0	0	7	<b>55</b>
Production Cell	32	3	6	0	8	0	1	4	0	4	<b>58</b>
Railroad Crossing	0	7	0	1	0	0	2	0	1	0	<b>11</b>
Stoplight	17	4	0	0	2	0	0	0	0	0	<b>23</b>
<b>Total</b>	<b>107</b>	<b>31</b>	<b>16</b>	<b>1</b>	<b>15</b>	<b>6</b>	<b>26</b>	<b>4</b>	<b>1</b>	<b>15</b>	<b>222</b>

In ASTRAL, the enablement of a transition depends on four factors: the local state, the imported state, the external environment, and the current time. The local state includes the values of local variables and the start and end times of local transitions. The imported state includes the values of variables imported from other processes and the start and end times of imported transitions. All transitions depend on one or more of these factors. For example, consider the two transitions of the bakery algorithm specification shown below.

```

TRANSITION set_choose
  ENTRY [TIME: exec_time]
    now ≥ delay
    & ~choosing
    & number = 0
  EXIT
    choosing

TRANSITION for_loop
  ENTRY [TIME: exec_time]
    next_i ≤ n_procs
    & ~choosing
    & number ≠ 0
    & ~procs[next_i].choosing
    & ( procs[next_i].number = 0
      | number < procs[next_i].number
      | number = procs[next_i].number
    & FORALL j: procs_int
      ( procs[j] = self
        → j ≤ next_i))
  EXIT
    next_i = next_i' + 1

```

The set\_choose transition depends on the local state (choosing and number) and the current time (Now). The for\_loop transition also depends on the local state (next\_i, choosing, and number), but in place of the current time, it depends on the other processes in the system (procs[next\_i].number) instead.

A transition is classified based on which factors its enablement is dependent. There are seven classifications corresponding to the possible combinations of the imported state (O), external environment (E), and current time (T) factors plus a classification for transitions that only depend on the local state (L). The set\_choose transition is of type T and the for\_loop transition is of type O. Table 3 shows the number of transitions of each classification that appear in the testbed systems. This information was obtained automatically using the transition classification

component of the ASTRAL SDE, which examines each transition for the different factors.

These classifications are significant because they can be used to help determine the delay between any two consecutive transitions on the same process instance, which in turn can be used to determine the execution time of arbitrary transition sequences. To determine the delay between two consecutive transitions tr1 and tr2, tr2 is first classified. If tr2 depends only on local variables, tr2 must fire immediately after tr1. If this were not the case, then a contradiction can be achieved as follows. Let t1 be the time that tr1 fired and t2 be the time that tr2 fired. In this case,  $t2 - t1 > \text{Duration}(\text{tr1})$  and tr2 is not enabled at  $t1 + \text{Duration}(\text{tr1})$  or else by the semantics of ASTRAL, tr2 would have fired. Since tr2 does eventually fire at t2, however, it is enabled at t2. Since tr2 depends only on the local state, there must have been a change to the local state between  $t1 + \text{Duration}(\text{tr1})$  and t2. This means that there was either a start, an end, and/or a change to a local variable. Since tr1 and tr2 were assumed to be consecutive, however, no other transition could have started or ended in between, and as a consequence, no changes to local variables could have occurred. This is a contradiction, so tr2 must fire immediately after tr1.

If tr2 depends on more than just the local state, the delay between tr1 and tr2 is more difficult to determine. If tr2 additionally depends on the current time, the history of the system must be examined to determine if and when the events constrained by the current time have occurred. In many cases, the events will be related to the execution of tr1. For example, a common instance of this type is “now - End(tr1) ≥ delay1”. Another common instance is “now - Change(v) ≥ delay1” where v is a variable set by tr1. In these cases, the delay between tr1 and tr2 is equal to delay1. If the events are not related to tr1, however, more in-depth analysis of the history of the system must be performed.

If tr2 depends on the environment, the start of tr2 is delayed until a call to tr2 is generated by the external environment. In this case, it is necessary to examine the environment clause for restrictions on calls to tr2. If there are no such restrictions, then the delay between tr1 and tr2 can be arbitrarily large. If there are restrictions, however, then it is necessary to examine the history of the system to determine when calls can or will occur to compute the delay between tr1 and tr2. Similarly, if tr2 depends on other processes, the start of tr2 is delayed until imported variables have the appropriate values and/or imported transitions have occurred at the appropriate times. In this case, the imported variable clause must be examined in a similar fashion to that of the environment clause. If tr2 depends on combinations of the

**Table 3. Transition classifications of testbed systems**

System	L	E	O	T	EO	ET	OT	EOT	Total
Bakery Algorithm	4	0	1	1	0	0	0	0	6
Cruise Control	2	9	2	1	0	0	0	0	14
Elevator	0	3	4	3	0	0	2	0	12
Olympic Boxing	0	0	2	2	0	0	1	1	6
Phone	0	2	16	0	7	0	5	0	30
Production Cell	14	0	11	20	0	0	10	1	56
Railroad Crossing	0	1	2	3	0	0	0	0	6
Stoplight	0	2	4	0	0	0	18	0	24
<b>Total</b>	<b>20</b>	<b>17</b>	<b>42</b>	<b>30</b>	<b>7</b>	<b>0</b>	<b>36</b>	<b>2</b>	<b>154</b>

current time, the environment, and other processes, all of the appropriate clauses must be examined.

## 6. RELATED WORK

Dwyer et al. [5] developed a set of property classifications based on a large number of linear temporal logic specifications collected from the literature. The classifications include absence, existence, universality, precedence, and response. It was found that almost all of the property specifications collected fall into one of these classifications. The goal of these classifications is to facilitate the creation and reuse of property specifications by providing a set of templates that can be instantiated appropriately during specification. This is in contrast to the property classifications presented in this paper, which were designed to be automatically recognizable after a specification has already been written to differentiate between distinct proof patterns. The two approaches, however, are not mutually exclusive. In fact, they can be used in a complementary fashion. For writing specifications, the template classification schemes can be used, which were developed with creation and reuse in mind, but were not developed to be statically recognizable or to distinguish between distinct proof styles. For analyzing specifications, the classification schemes of this paper can be used, which are statically recognizable and distinguish between distinct proof styles, but may not be intuitive or specific enough to be used as design templates.

In a similar fashion, Shaw and Garlan [19] developed a taxonomy where systems are classified based on different types of system components and the read/write interactions between them. Classifications in the taxonomy include pipes and filters, object-orientation, implicit invocation, layered systems, repositories, interpreters, and process control. The purpose of these classifications is to facilitate the design and reuse of software architectures by identifying the most common patterns that occur in practice. This classification scheme is supported by an automatic classification tool developed by Dean and Cordy [4]. A system is first specified as a graph, where nodes represent different components in the system and edges represent different types of connections between components. Node types include tasks, tables, random access repositories, and files. Edge types include streams, memory accesses, messages, procedures, invocations, and productions. Once the graph for a system has been specified, a pattern-matching algorithm determines the type of the system based on the Shaw/Garlan taxonomy. Although these classifications can be automatically identified, they do not differentiate enough between distinct proof styles. In other words, systems that have identical classifications may need significantly different tools and techniques to discharge their proofs.

Henzinger et al. [8] provide guidance as to how proofs can be constructed. In this work, proof methodologies based on temporal logic reasoning are discussed for timed transition systems. Two different specification styles are identified and different proof techniques are presented for each. In the first specification style, real-time properties are expressed using time-bounded temporal operators. Proof rules are provided for proving bounded-invariance and bounded-response properties. In the second specification style, real-time properties are expressed using an explicit clock variable. Since the clock variable can be thought of as just an ordinary variable, untimed proof techniques are used to discharge proofs in these systems. These classifications are at a

much higher level than the classifications of this paper and cannot be used to direct the proofs at the level needed by a human prover.

## 7. CONCLUSIONS AND FUTURE WORK

This paper has presented process, property, and transition classification schemes for real-time systems that can be used to assist in the analysis of these systems. By basing the classification schemes on actual specifications and their associated proofs, the likelihood of getting guidelines and tools that will help the user in analyzing and proving other specifications is increased. A large number of the proofs of the testbed systems were proved by hand with the assistance of the classification schemes and proof techniques, which became available incrementally as more and more proofs were performed. Over half of the requirements were completely proved in PVS. A systematic methodology for the analysis of real-time systems based on these classification schemes can be found in [11].

In general, more example systems need to be specified and proved to determine whether the classification schemes presented in this paper are sufficient to classify all real-world systems or whether other useful classifications exist. Although the classification schemes developed were based on ASTRAL specifications, most of the descriptions did not rely on exclusively ASTRAL concepts and terminology, but instead were discussed in a high level fashion. Thus, the authors feel that these schemes are also applicable to other real-time concurrent specification languages. For languages similar to ASTRAL, such as the Timed Automaton Model [15] and the Timed Transition Model [18], these classifications are readily applicable. For other languages, the classification schemes can be applied by taking differences in language constructs into account. Process classification corresponds to the classification of the active entities of the language, whether they be automata, objects, etc. Property classification is similar across most real-time languages with slight differences for the logic used and the operators available. Finally, transition classification corresponds to the classification of the state changes of a language, whether they be transitions, actions, functions, etc. Further investigation is necessary to validate that the classifications are applicable in general.

The classification schemes discussed in this paper have been incorporated into the ASTRAL SDE. Queries to display process and transition classification information are available in their respective browsers, and the property classification information is automatically displayed whenever the formula splitter tool is invoked on a formula. In addition, fully automated theorem prover strategies are available for properties classified as untimed properties [10]. Finally, the transition classification information is used by the transition sequence generator [10] to estimate the running times of each transition sequence that is generated.

## 8. ACKNOWLEDGMENTS

This research was partially supported by NSF Grant No. CCR-9204249.

## 9. REFERENCES

- [1] Borgida, A., J. Mylopoulos, and R. Reiter, "On the Frame Problem in Procedure Specifications", *IEEE Transactions on Software Engineering*, 21(10), 785-198 (Oct., 1995).

- [2] Coen-Porisini, A., C. Ghezzi, and R.A. Kemmerer, "Specification of Realtime Systems Using ASTRAL", *IEEE Transactions on Software Engineering*, 23(9), 572-598 (Sept., 1997).
- [3] Crow, J., S. Owre, J. Rushby, N. Shankar, and M. Srivas, "A Tutorial Introduction to PVS", *Proc. Workshop on Industrial-Strength Formal Specification Techniques*, (Apr., 1995).
- [4] Dean, T.R. and J.R. Cordy, "A Syntactic Theory of Software Architecture", *IEEE Transactions on Software Engineering*, 21(4), 302-333 (Apr., 1995).
- [5] Dwyer, M.B., G.S. Avrunin, and J.C. Corbett, "Patterns in Property Specifications for Finite-State Verification", *Proc. 21st Int. Conference on Software Engineering*, 411-420, (May, 1999).
- [6] Filman, R.E. and D.P. Friedman, *Coordinated Computing: Tools and Techniques for Distributed Software*, McGraw-Hill, New York, NY, (1984).
- [7] Heitmeyer, C. and N. Lynch, "The Generalized Railroad Crossing: a Case Study in Formal Verification of Real-Time Systems", *Proc. 15th Real-Time Systems Symposium*, 120-131 (Dec., 1994).
- [8] Henzinger, T.A., Z. Manna, and A. Pnueli, "Temporal Proof Methodologies for Timed Transition Systems", *Information and Computation*, 112(2), 273-337 (Aug., 1994).
- [9] Kolano, P.Z., "The ASTRAL Specifications of 8 Real-Time Systems", *Technical Report TRCS99-08*, Computer Science Department, University of California, Santa Barbara, CA (Mar., 1999).
- [10] Kolano, P.Z., "Proof Assistance for Real-Time Systems Using an Interactive Theorem Prover", *Proc. 5th Int. AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems*, 315-333 (May, 1999).
- [11] Kolano, P.Z., "Tools and Techniques for the Design and Systematic Analysis of Real-Time Systems", Ph.D. Thesis, University of California, Santa Barbara, CA (Dec., 1999).
- [12] Kolano, P.Z., Z. Dang, and R.A. Kemmerer, "The Design and Analysis of Real-Time Systems Using the ASTRAL Software Development Environment", *Annals of Software Engineering*, 7, 177-210 (1999).
- [13] Lamport, L., "A New Solution of Dijkstra's Concurrent Programming Problem", *Communications of the ACM*, 17(8), 453-455 (Aug., 1974).
- [14] Lewerentz, C. and T. Lindner, eds., *Formal Development of Reactive Systems: Case Study Production Cell*, Springer-Verlag, New York, NY (1995).
- [15] Lynch, N. and F. Vaandrager, "Forward and Backward Simulations for Timing-Based Systems", *Proc. REX Workshop on Real-Time Theory in Practice*, 397-446 (June, 1991).
- [16] Manna, Z. and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, New York, NY (1992).
- [17] Official 1996 Olympic Web Site, <http://www.atlanta.olympic.org> (1996).
- [18] Ostroff, J.S., *Temporal Logic for Real-Time Systems*, Research Studies Press, Taunton, UK (1989).
- [19] Shaw, M. and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, Upper Saddle River, NJ (1996).
- [20] Ward, P.T. and S.J. Mellor. *Structured Development for Real-Time Systems*, Yourdon Press, New York, NY (1985).