Verification of Timing Properties in Rapid System Prototyping*

Doron Drusinky and Man-Tak Shing
Department of Computer Science
Naval Postgraduate School
833 Dyer Road
Monterey, CA 93943 USA
{drusinsky, shing}@nps.navy.mil

Abstract

This paper addresses the need for systematic verification of timing properties of real-time prototypes, which consist of timing constraints that must be satisfied at any given time and time-series constraints that must be satisfied over a period of time. Traditional schedulability analysis only works for the former kind of timing properties. It is not effective in verifying time-series constraints over a period of time. This paper presents a hybrid approach that combines the traditional schedulability analysis of the design and the monitoring of timing constraint satisfaction during prototype execution based on a time-series temporal logic. The effectiveness of the approach is demonstrated with a prototype of the fish farm control system software.

1 Introduction

Real-time systems are those whose correct behavior depends not only on the logical result of the computation but also on the time at which the result is produced. Traditionally, these temporal requirements are expressed as hard and soft timing constraints. It is imperative for real-time systems to meet all deadlines in hard timing constraints but acceptable to miss the deadlines of the soft timing constraints occasionally [1]. There are currently two complementary approaches to evaluating the correctness of real-time systems: static analysis of its behavior according to a set of metrics (e.g. schedulability analysis to establish the feasibility of the timing constraints) and run-time monitoring of real-time systems to study its behavior according to set of metrics (e.g. release jitter, frequency and degree of tardiness, etc.). While the static analytic approach plays a very important role in helping system designers set time budgets and allocate resources in their designs, they are only effective if correct timing constraints can be determined during the requirements analysis phase. Feasible requirements for large dynamic systems are difficult to formulate, understand, and meet without extensive prototyping. Moveover, traditional analytical techniques are not effective in evaluating time-series temporal behaviors (e.g. the maximum duration between consecutive missing deadlines must greater than 5 seconds). This kind of requirements can only be evaluated through execution of the real-time systems or their prototypes.

Computer aid holds the key to rapid construction, evaluation, and evolution of such prototypes. Analysis and measurement of prototype designs provides upper bounds on the time required to execute particular functions, and experiments with simulated environments provide information about the accuracies and response times required to keep external physical systems within desired operating regions.

This paper presents hybrid approach that combines the static schedulability analysis of the design and the run-time monitoring of the prototype execution based on a time-series temporal logic. The approach is supported by an environment made up of the Software Engineering Automation Tools (SEATools) [2] and the DBRover System [3]. The effectiveness of the approach is demonstrated with a prototype of the fish farm control system software.

2 Models and Tools for Analysis, Design and Prototyping of Complex Systems

We build upon our experience with prototyping languages [4], real-time systems modeling and analysis [5, 6], automatic code generation techniques and rapid prototyping environments [7] and developed a set of PC-based com-

* This research was supported in part by the U.S. Army Research Office under grant number 40473-MA-SP.

puter-aided tools to support the modeling, analysis and prototyping of the systems under development (Figure 1).

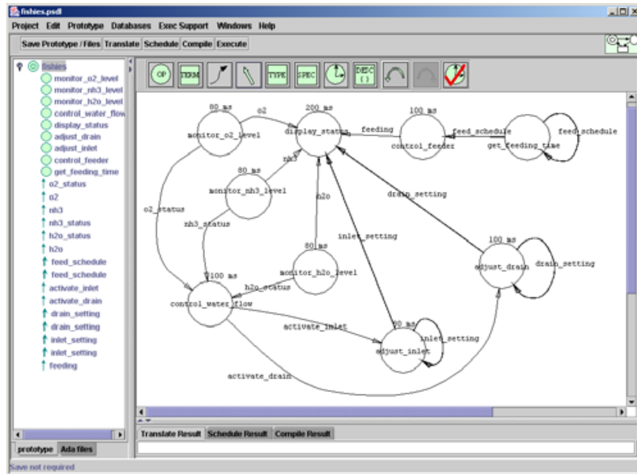

Figure 1. The SEATools Environment

The SEATools is based on the Prototyping System Description Language (PSDL) [4], which is a high-level language designed specifically to support the conceptual modeling of real-time embedded systems. Real-time requirements in the system development are modeled as PSDL specifications, which are dataflow graphs augmented with non-procedural timing and control constraints (Figure 2). PSDL allows the specification of both input and output guards to provide conditional execution of an operator and conditional output of data. Guards can include conditions on timers that measure duration of system states, and can allow operators to execute only when fresh data has been written to an input stream. Each time critical operator has a *maximum execution time (MET)* constraint, representing the maximum time the operator may need to complete execution after it is fired, given access to all required resources. In addition, each periodic operator has a *period* and a *deadline (FW)*. The period is the interval between triggering times for the operator and the deadline is the maximum duration from the triggering of the operator to the completion of its operation. Each sporadic operator has a *maximum response time (MRT)* and a *minimum calling period (MCP)*. The minimum calling period is the smallest interval allowed between two successive triggering of a sporadic operator. The maximum response time is the maximum duration allowed from the triggering of the sporadic operator to the completion of its operation.

PSDL's declarative timing and control constraints help decouple the behavioral aspects of a system from its timing properties to allow independent analysis of these two aspects, and organize timing constraints in a hierarchical fashion, to allow independent consideration of smaller subsets of timing constraints.

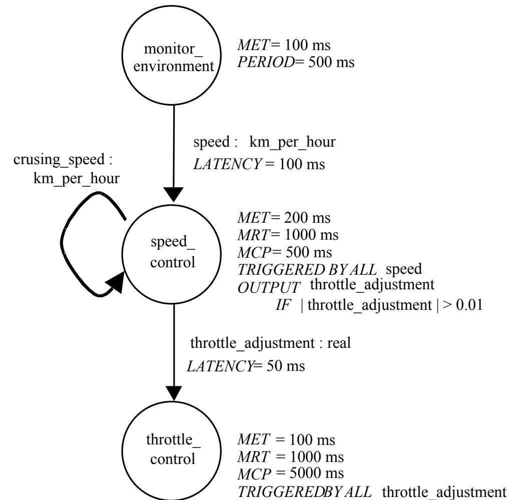

Figure 2. PSDL specification

The SEATools User Interface provides the essential facilities for users to create and modify the models. It also provides some degree of computer-aided consistency checking and data entry propagation at the user interface level. Complete semantic check of the model and static analysis of the timing constraints are performed by the SEATools' execution support system, which consists of a translator, a scheduler and a runtime monitor. The translator checks the semantics of the model and generates code that binds together the code supplied by the designer and the reusable components extracted from the software base. The scheduler analyzes the feasibility of the timing constraints against the resources specified in the target hardware model and create the real-time schedule and control code needed for executing the prototype. The resultant Ada main program consists of four parts. The first is a set of data streams, implemented as instantiation of generic packages containing Ada tasks controlling the mutually exclusive read/write access to the data buffers. The second part consists of a set of drivers, one for each of the atomic operators. Each driver reads data from the specified input streams, evaluates the input guards, executes the operators, evaluates the output guards and then writes the outputs to the specified data streams accordingly. The third part is a static schedule, which is a high priority Ada task that executes all time-critical operators in a deterministic and timely manner. The schedule is generated automatically based on the timing constraints and the precedence of the operators specified in the data-flow graph. The last part of the Ada main program is a dynamic schedule, which is a low priority Ada task that executes the non-time-critical operators during the slack time in the static schedule. SEATools also includes a simple runtime monitor that checks for missing deadlines during prototype execution.

3 Metric Temporal Logic with Time Series Constraints

Temporal Logic is a special branch of modal logic that investigates the notion of time and order. In [8], Pnueli suggested using Linear-Time Propositional Temporal Logic (LTL) for reasoning about concurrent programs. Since then, several researchers have used LTL to state and prove correctness of concurrent programs, protocols, and hardware (e.g., [9, 10]).

Linear-Time Temporal Logic (LTL) is an extension of propositional logic where, in addition to the propositional logic operators there are four future-time operators and four dual past time operators: a*lways* in the future (always in the past), *eventually*, or sometime in the future (sometime in the past), *Until* (Since), and *next* cycle (previous cycle).

Metric Temporal Logic (MTL) was suggested by Chang, Pnueli, and Manna as a vehicle for the verification of real time systems [11]. MTL extends LTL by supporting the specification of relative time and real time constraints. All four LTL future time operators (*Eventually*, *Always*, *Until*, *Next*) can be characterized by relative time and real time constraints specifying the duration of the temporal operator. Hence, for example, the MTL assertion "Always < 20 commandResult > 0", states that commandResult > 0 must hold every cycle until 20 cycles in the future.

MTL with time-series constraints (MTLS) enables the specification of requirements in which propositions include temporal instances of variables. Consider the following *automotive cruise control* code with a stability assertion (using embedded TemporalRover syntax [3]) requiring speed to be 5% stable while cruise is set and not changed:

```
void cruise(boolean cruiseSet, boolean
      cruiseChange,
      boolean cruiseOff, boolean
      cruiseIncr, int speed){
… /* Cruise Controller functionality */
/* TRBegin
TRAssert{Always ({cruiseSet}Implies
    {speed*0.95 < speed’ &&
     speed’ < speed*1.05}
         Until $speed$
    {cruiseChange || cruiseOff})}=>
                 {…} // user actions
TREnd */
```

In the example *speed* is a temporal data variable, which is associated with the *Until* temporal operator. This association implies that every time the *Until* operator begins its evaluation, possibly in multiple instances (due to non-determinism), the speed value is sampled and preserved in *speed* variable of this instance of the *Until*; this value is referred to as the *pivot* value for this *Until* node instance. Future speed values used by this particular evaluation of the Until statement are referred to using the *prime* notation, i.e., as *speed'*; these future instances of the speed variable are referred to as *primed values*. Hence, if the speed value was 100Km/h when *cruiseSet* is true, then the pivot value for speed is 100, while every subsequent speed is referred to as *speed'* and must be within 5% of the pivot speed value.

Note how speed is *declared* using the *$speed$* notation to be a temporal data variable associated with the Until operator. This declaration indicates to the Temporal Rover that it should be sampling a pivot value from the environment in the first cycle of the Until operators lifecycle, and to refer to all subsequent samples of speed as *speed'*.

Similarly, the following example consists of a *monotonicity* requirement for the cruise control system, where speed is monotonically increasing while Cruise Increase (*cruiseIncr*) command is active:

```
TRAssert {Always({cruiseIncr}Implies
   {(speed<=speed')&& speed=speed')>=0}
       Until    $speed$    {!cruiseIncr}
  )}=>  {…} // user actions
```

In this example the temporal data variable *speed* is sampled upon every *cruiseIncr* event , and is compared to the current value (*speed'*) every cycle. The latest speed value is then saved in the pivot for next cycles comparison.

4 The DBRover Run-Time Monitor

The DBRover is an MTLS monitoring tool based on the TemporalRover code generator of [3]. It consists of a GUI for editing temporal assertions, an MTLS simulator, and an MTLS execution engine (Figure 3).

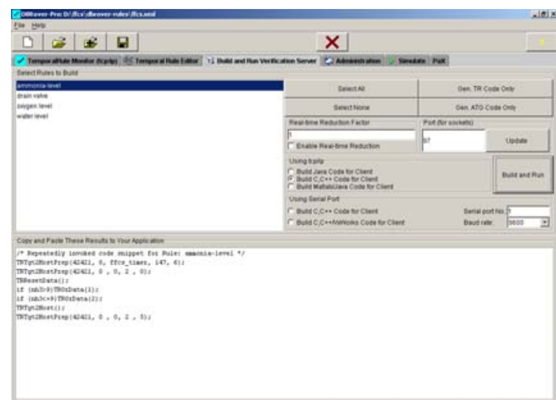

Figure 3. The DBRover System

The DBRover builds and executes temporal rules for a target program or application. In run-time, the DBRover lis-

tens for messages from the target application and evaluates corresponding temporal assertions. Hence, in the cruise-control example above, the DBRover will listen for Boolean messages pertaining to the run-time values of the `cruiseSet`, `cruiseChange`, and `cruiseOff` Boolean propositions, as well as the run-time value of the `speed` variable. The DBRover then evaluates the corresponding MTLS assertion for that cycle. Monitoring is performed *on-line*, namely, the DBRover operates in tandem with the target program, and re-evaluates assertions every cycle. The DBRover uses an underlying algorithm that does not store a history trace of the data it receives; it can therefore monitor very long, and potentially never ending, executions of target applications.

5 Prototype Generation and Runtime Verification of the timing properties

In this section, we shall illustrate the hybrid approach with a fish farm control system prototype.

5.1 The Fish Farm Control System (FFCS)

The FFCS will control the fish food dispenser and water quality in a fish tank. The tank has a mechanical feeder that drops pellets of fish food from a feeder tube suspended above the tank. The feeder can be turned on and off by the computer. The tank also has a water inlet pipe and a drain pipe with valves controlled by the computer, and sensors that measure the water level (millimeters above the bottom), the oxygen level in the water (parts per million), and the ammonia level in the water (parts per million).

The FFCS must deliver fish food at scheduled feeding times, repeated every day. The times when each feeding starts and stops are displayed on the console of the FFCS and can be adjusted from the keyboard.

The FFCS must keep the oxygen level at least 8 parts-per-million (ppm), and the ammonia level at most 9 ppm. Fish will die if left in an environment with low oxygen or high ammonia for 1 minute or more. The fish tank is 1 meter wide, 2 meters long, and 1 meter deep (1mm level = 2 liters volume). The FFCS must keep the water level between 60 and 90 cm at all time. The fill/drain valves allow a maximum flow of 0.5 liters per second when valve is fully open. The fresh water coming in the inlet valve contains 30 ppm of oxygen and contains no ammonia. The fish in the tank consumes oxygen at a rate of 0.1 ml/sec and generates ammonia at a rate of 0.0015ml/sec while resting and at a rate of 0.003 ml/sec while they're eating.

The FFCS should minimize water flow subject to the above constraints.

In addition, we add another requirement that "whenever water level is below 88 cm for at least three minutes, the drain valve settings should be limited to be at most 10% of the maximum setting per second" to illustrate the expressive power of the temporal logic.

5.2 The PSDL Model

Figure 4 shows the PSDL model for the FFCS. In the interest of brevity, we shall only discuss the water quality control portion of the prototype in this paper, which is made up of six atomic operators: *monitor_h2o, monitor_o2, monitor_nh3, control_water_flow, adjust_inlet and adjust_drain,* with the associated control and timing constraints shown in Table 1.

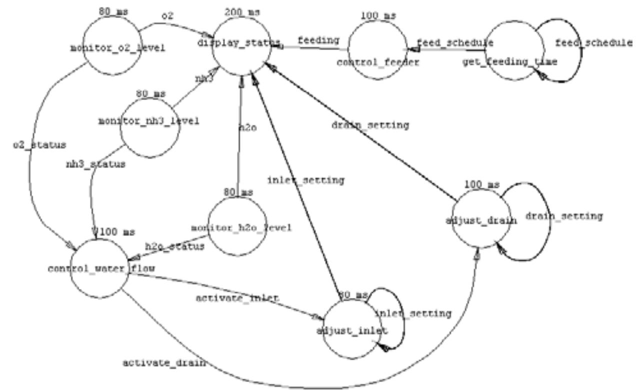

Figure 4. The PSDL model for the Fish Farm Control System

Operator	Control Constraints	Timing Constraints
monitor_h2o	–	Period = 2000 ms FW = 200 ms MET = 80 ms
monitor_o2	–	Period = 2000 ms FW = 200 ms MET = 80 ms
monitor_nh3	–	Period = 2000 ms FW = 200 ms MET = 80 ms
control_water_flow	–	Period = 1000 ms FW = 200 ms MET = 100 ms
adjust_inlet	Triggered by SOME activate_inlet	MCP = 2000 ms MRT = 2500 ms MET = 80 ms
adjust_drain	Triggered by SOME activate_drain	MCP = 2000 ms MRT = 2500 ms MET = 80 ms

Table 1. The control and timing constraints of the water quality control operators

Central to the design is the *control_water_flow* operator, which controls the inlet and drain water flow based on the following decision table.

Water Level	< 65 cm	≥ 65 cm, ≤ 85 cm		> 85 cm	
Oxygen (O2) & Ammonia (NH3) Level	–	O2 < 8 ppm or NH3 > 9 ppm	O2 ≥ 8 ppm and NH3 ≤ 9 ppm	O2 < 8 ppm or NH3 > 9 ppm	O2 ≥ 8 ppm or NH3 ≤ 9 ppm
Inlet Valve Setting	open	open	close	open	close
Drain Valve Setting	close	close	close	open	open

Table 2. Decision table for the control water flow logic

5.3 Incorporating DBRover Run-time Checking to the PSDL Model

Next, we want to find out if the prototype meets all the requirements using the DBRover System. We use the TemporalRover to generate C code for the following temporal rules:

Rule 1: The water level must be between 60 and 90 cm at all time, formally written as:

Always {h2o >= 60 && h2o <=90}.

Rule 2: The oxygen level cannot be less than 8 ppm for more than 60 seconds, formally written as:

Always {o2<8} Implies Eventually $_{<=60}$ *{o2>=8}.*

Rule 3: The ammonia level cannot be more than 9 ppm for more than 60 seconds, formally written as:

Always {nh3>9} Implies Eventually $_{<=60}$ *{nh3<=9}.*

Rule 4: If water level has been below 88 cm for 180 seconds, then the change of the drain valve setting must be less than or equal to 10% of the maximum setting per second (100), formally written as:

Always(Always $_{>=180}$ *{h2o<=88} Implies Eventually $dv, ffcs_timer$ {abs(dv' – dv)/(ffcs_timer'-ffcs_timer) <= 10}.*

We also add four operators (*check_h2o_level, check_o2_level, check_nh3_level, check_drain_setting*) to the PDSL model (Figure 5). These operators, when triggered respectively by new data values in the *h20, o2, nh3* and *drain_setting* streams, will send the updated values to the DBRover for temporal property verification during prototype execution (Figure 6). The control and timing constraints of these operators are shown in Table 3.

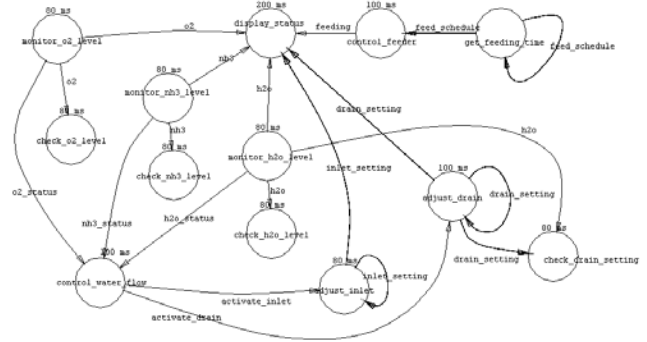

Figure 5. The enhanced PSDL model with additional operators to invoke the DBRover runtime monitor

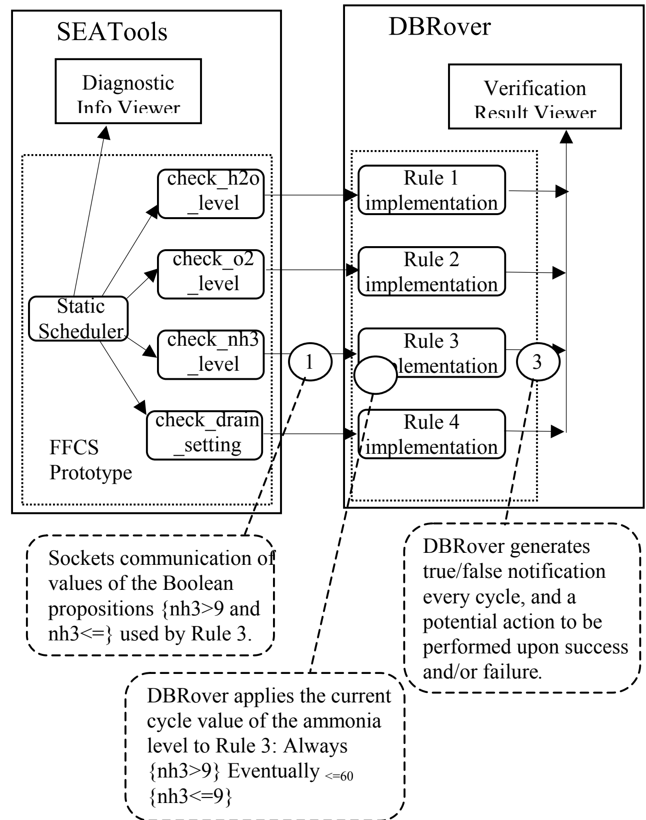

Figure 6. Architecture of the integrated SEATools / DBRover Runtime Monitor System.

Operator	Control Constraints	Timing Constraints
check_h2o_level	Triggered by SOME h2o	MCP = 1000 ms MRT = 1500 ms MET = 80 ms
check_o2_level	Triggered by SOME o2	MCP = 1000 ms MRT = 1500 ms MET = 80 ms
check_nh3_level	Triggered by SOME nh3	MCP = 1000 ms MRT = 1500 ms MET = 80 ms
check_drain_setting	Triggered by SOME h2o, drain_setting	MCP = 2000 ms MRT = 2500 ms MET = 80 ms

Table 3. The control and timing of the new operators

6 Conclusions

Traditionally, rapid prototyping and formal or run-time verification methods have been applied to, and thought of as, two separate phases of the design process. Rapid prototyping has traditionally been used in early stages of the design process, for the purpose of early system evaluation and demonstration, prior to implementation and coding. In contrast, formal and run-time verification methods have been used in later stages of the design process, to validate and debug code that has already been written.

This paper shows that run-time monitoring and verification can be applied much earlier in the design process, in tandem with rapid prototyping. This approach helps identify errors earlier in the design process and also helps debug the requirements themselves.

The executable prototype consists 3968 lines of source code, 2048 of which are Ada and C codes generated by the SEATools and the TemporalRover. The use of socket communication provides a very simple interface between the SEATools runtime environment and the DBRover System. We only need to create one atomic operator in the PSDL model for each temporal rule. The Ada implementation of each of these atomic operators consists of a one-line procedure call in the to invoke the corresponding C routine implementing the temporal rule. The mapping between the Ada and C code is very straightforward and can be automatically generated easily. Although the use of socket communication introduces additional time delay between the detection of events during the prototype execution and the checking of the affected temporal properties by the DBRover, it has negligible effect on the accuracy of the verification result because DBRover allows user to specify time based on the client's clock. All events detected during prototype execution are stamped with the local clock before sending to the DBRover for verification.

References

[1] J. Liu, *Real-Time Systems*, Prentice Hall, 2000.

[2] Luqi, et al., "SEA Environment for CARA Software", in the Tech. Report NPS-SW-03-001, Naval Postgraduate School, Monterey, CA, 2003, pp. 169-196.

[3] D. Drusinsky, "*The Temporal Rover and ATG Rover", Proc. Spin2000 Workshop, Springer Lecture Notes in Computer Science*, 1885, pp. 323-329.

[4] Luqi, V. Berzins, and R. Yeh, "A Prototyping Language for Real-Time Software", *IEEE Trans. on Software Eng*., 14(10), pp. 1409-1423 (1988).

[5] Luqi, "Real-Time Constraints in a Rapid Prototyping Language", *Journal of Computer Languages*, 18, pp. 77-103 (1993).

[6] Luqi and M. Shing, "Real-Time Scheduling for Software Prototyping", *Journal of Systems Integration, Special Issue on Computer Aided Prototyping*, 6, pp. 44-72 (1996).

[7] Luqi et. al., "DCAPS-architecture for distributed computer aided prototyping system", *Proc.* *12th International Workshop on Rapid System Prototyping*, 2001, pp. 103 –108.

[8] A. Pnueli, "The Temporal Logic of Programs", *Proc. 181977 IEEE Symp. on Foundations of Computer Science*, pp. 46-57.

[9] B. T. Hailpern and S. Owicki, "*Modular Verification of Communication Protocols", IEEE Trans of comm*., COM-31(1), No. 1, 1983, pp. 56-68.

[10] Z. Manna and A. Pnueli, "Verification of Concurrent Programs: Temporal Proof Principles", *Proc. of the Workshop on Logics of Programs, Springer LNCS*, 1981 pp. 200-252.

[11] E. Chang, A. Pnueli and Z. Manna, "Compositional Verification of Real-Time Systems", *Proc. 9th IEEE Symp. On Logic In Computer Science*, 1994, pp. 458-465.