Test-time, Run-time, and Simulation-time Temporal Assertions in RSP

Doron Drusinsky, Man-Tak Shing and Kadir Demir
Department of Computer Science
Naval Postgraduate School
833 Dyer Road, Monterey, CA 93943, USA
{ddrusins, shing, kdemir}@nps.edu

Abstract

For cost-effective prototyping, system designers should have a clear understanding of the intended use of the prototype under development. This paper describes a classification of formal specification (temporal) assertions used during system prototyping. The classification introduces two new classes of assertions in addition to the well-known class of test-time assertions: (i) assertions used only during simulation, and (ii) deployable assertions integrated with run-time control flow. Separating the formal specification into three distinct classes allows system designers to develop more effective prototypes to evaluate the different system behaviors and constraints. A prototype of a naval torpedo system is used to illustrate the concept.

1 Introduction

The analysis and design of complex safety-critical embedded systems pose many challenges. Feasible timing and safety requirements for these systems are difficult to formulate, understand, and meet without extensive prototyping. Traditional timing analysis techniques are not effective in evaluating time-series temporal behaviors (e.g. the maximum duration between consecutive missed deadlines must be greater than 5 seconds). This kind of requirements can only be evaluated through execution of the real-time systems or their prototypes. Rapid prototyping also helps system designers formulate and evaluate safety requirements of the system under development, by building two separate models (one for the system under development and the other for the environment (or equipment) under its control) and then exercising the two models in tandem to see if the simulation ends up in known hazardous states under normal operating conditions and under various failure conditions [AL].

Run-time Execution Monitoring of formal specification assertions (REM) is class of methods of tracking the temporal behavior of an underlying application. REM methods range from simple print-statement logging methods to run-time tracking of complex formal requirements (e.g., written in temporal logic) for verification purposes [D3]. Recently, NASA used REM for the verification of flight code for the Deep Impact project [DW]. Also recently, the U.S. Ballistic Missile Defense System has adopted REM as the primary verification method for the new ballistic missile battle manager because of its ability to scale, and its support for temporal assertions that include real-time and time series constraints [CDMSS]. In [DS], we showed that the use of run-time monitoring and verification of temporal assertions, in tandem with rapid prototyping, helps debug the requirements and identify errors earlier in the design process. For cost-effective prototyping, the system designers should have a clear understanding of the intended use of the prototype under development.

Published REM methods typically use temporal logic as a specification language [D2, HR]. Traditionally, run-time verification methods have been used in later stages of the design process, to validate and debug code that has already been written. Correctness assertions of interest were created for the purpose of REM-based testing or for model checking, in other words they were test-time assertions. Based on our previous research on the use of REM in rapid prototyping via modeling and simulation [DM] and in fortifying target software's exception-handling ability [D1], we present in this paper two additional classes of assertions: (i) assertions that are used only during simulation, and (ii) deployable assertions integrated with the run-time control flow of the target software. Separating the formal specification into three distinct types allows system designers to develop more effective prototypes to evaluate the different system behaviors and constraints. We will illustrate these different classes of assertions and their intended use with an example prototype of a naval torpedo system using the OMNeT++ prototyping environment [Va] and the DBRover REM tool [D3].

The rest of the paper is organized as follows. Section 2 provides a short introduction to temporal logic. Section 3 gives an overview of an autonomous homing torpedo control software prototype developed using the OMNeT++ tools. Section 4 illustrates the different kinds of assertions using torpedo control software example and discusses their impli-

Proceedings of the 16th International Workshop on Rapid System Prototyping (RSP'05)
1074-6005/05 $20.00 © 2005 IEEE

cations on the prototype design. Section 5 shows how these assertions can be evaluated using the OMNeT++ prototyping environment and the DBRover REM tool, and section 6 draws the conclusion.

2 Temporal Logic

Temporal Logic is a special branch of modal logic that investigates the notion of time and order. Linear-time Temporal Logic (LTL) is an extension of propositional logic where, in addition to the well-known propositional logic operators, there are four future-time operators (◊-*Eventually*, □-*Always*, *U-Until*, O-*Next*) and four dual-past time operators. Pnueli [Pn] suggested using LTL for reasoning about concurrent programs. Since then, several researchers have used LTL to state and measure correctness of concurrent programs, protocols, and hardware (e.g., [MP, Pn]). Metric Temporal Logic (MTL) was suggested by Chang, Pnueli, and Manna as a vehicle for the verification of real time systems [CP]. MTL extends LTL by supporting the specification of relative-time and real-time constraints. With MTL, all four LTL future-time operators can be characterized by relative-time and real-time constraints specifying the duration of the temporal operator. Temporal Logic with Time Series constraints (MTLS) was suggested by Drusinsky as an extension of MTL which enables temporal specifications that assert about time-series properties such as stability, monotonicity, and min-max values [D2]. For example, the following *automotive cruise control* code with a stability assertion (using embedded TemporalRover syntax [D3]) 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} =>
           {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 the *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.

3 Autonomous Torpedo System Prototype

We shall illustrate the concepts with the real-time control software of an autonomous homing torpedo, called KTorp. KTorp is a submarine-launched torpedo that searches, tracks and destroys its underwater targets. The torpedo consists of five main sections (Sonar, Warhead, Battery, Control and Engine) and can operate in any sea conditions. It has a maximum navigation time limit of 35 minutes at 30 knots, a navigation depth between 35 feet and 2500 feet, and two search modes. A torpedo run consists of three sequential phases: Enable, Search and Attack. The torpedo is in its Enable phase when it leaves its mother ship. The torpedo is disarmed and its sonar is inactive, so that it will not be able to attack its mother ship. The torpedo will follow a straight path until it reaches its enable distance. The torpedo then enters its Search phase and starts searching for its target with the predefined algorithm according to its search mode. When the torpedo finds its target, it enters its Attack phase and attacks with a predetermined specification according to its search mode.

3.1 KTorp Prototype

The torpedo prototype was created in two phases using the OMNeT++, an object-oriented discrete-event simulator primarily designed for the simulation of communication protocols, communication networks and traffic models, and models of multiprocessor and distributed systems. Phase I concentrates on the modeling of torpedo subsystem behavior while phase II expands and refines the environment model to capture the events and responses between the torpedo and its environment. Phase II is needed for the verification of the simulation-time and run-time assertions.

Figures 1 and 2 show the OMNeT++ model for the Phase I KTorp prototype and its User Interface for the simulation.

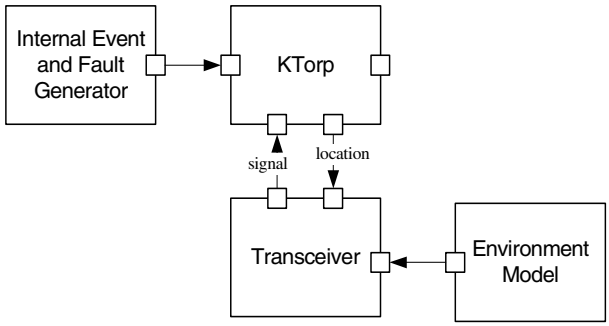

Figure 1. The Phase I KTorp OMNeT++ Model

Proceedings of the 16th International Workshop on Rapid System Prototyping (RSP'05)
1074-6005/05 $20.00 © 2005 **IEEE**

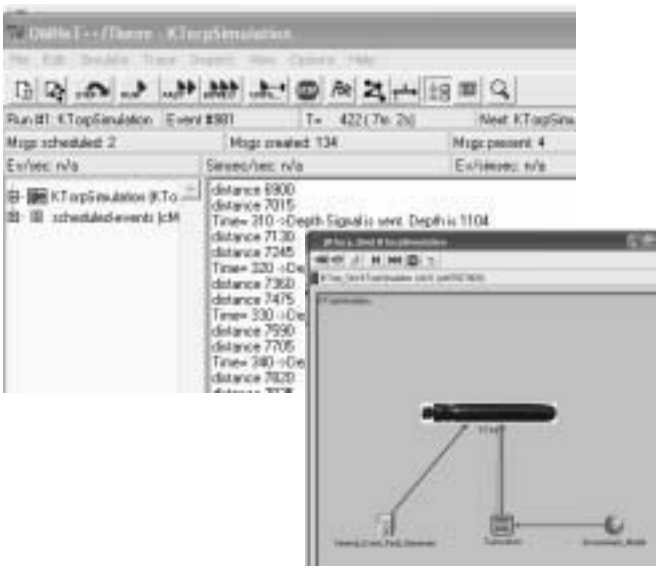

Figure 2. The OMNeT++ User Interface of the Phase I KTorp Simulation

The KTorp model is made up of four modules:

1. The *Environment Model* module provides a simplified model of the underwater environment, which is a complex, dynamic environment that varies according to the weather condition, seasons, and regional differences. We assume that the environment consists of two types of signals (target signals and false echo signals) that are detectable by the transceiver device of the torpedo. The target signals come from a real target and consist of multiple signals repeated at regular interval. The false echo signals, although very similar to the target signals, differ from the target signals in the number of signals and in the frequency they are generated.
2. The *Transceiver* module models the transceiver device situated in the front of the torpedo. We assume that the device is a passive device in this model, i.e. it only receives signals. To make the simulation more realistic, the device uses the location data from the *KTorp* module to filter out input signals that are more than 1000 yards away from the torpedo. This device has some precision and reliability. The reliability of the device will be modeled using probability based on the input parameters entered by the system designer at the beginning of a simulation run.
3. The *Internal Event and Fault Generator* module provides an abstraction of the sensors inside the torpedo: (i) a small device called the Enabler responsible for detecting that the torpedo is at the end of the Enable phase and sending an "Enable Signal" to the control logic to arm itself and begin its search for the target, and (ii) an Inertial Measuring Unit (IMU) responsible for measuring the distance that KTorp traveled. The Enabler uses the data generated by the IMU to decide if the torpedo reaches the enable distance from the mother ship.
4. The *KTorp* module models the control logic of the torpedo.

4 Test-time, Run-time, and Simulation-time Assertions in the Torpedo Prototype

Requirements analysts typically start their requirements discovery process based on some scenarios involving the system and its environment, and express their understanding of the expected behavior or properties of the system informally with natural languages. Mission-essential and safety critical functional and non-functional requirements are then expressed as formal assertions to improve the requirements' clarity and precision. System designers take the requirements (expressed either in natural languages or as formal assertions) and create design artifacts of a system meeting the requirements. They may annotate the design artifacts with some of the assertions obtained in the requirements phase and add new assertions to express the important properties specific to the system being built. When the programmers map the system design to code, they may embed some of the formal assertions in the design artifacts into the code and may add more assertions to specify the correct behavior of the code modules for test automation or run-time exception detection. Depending on the intended and potential use of these requirements, design and code assertions in REM, we can classify them into the following three different classes.

4.1 KTorp Safety Requirements and Test-time Assertions

It is important for safety-critical systems like KTorp to behave properly even under abnormal or erroneous conditions. Many of these safety-critical behaviors are expressed in formal assertions intended for testing the correctness of the design and/or implementation. The following are test-time assertions for the torpedo.

1. If the torpedo reaches its maximum navigation time limit (35 minutes) and still unable to find its target, the torpedo should never explode. Therefore the power to the warhead section must be cut and the torpedo will be disarmed. The MTL for this assertion is:

   Rule 1: $Always\ (launch \Rightarrow Eventually_{>=35\ min}(live \Rightarrow Eventually_{<2\ sec}\ disarmed))$.

2. The torpedo should never explode before it reaches its enable-distance from the mother ship (600 yards). Let *IMUdist* denote the distance traveled by the torpedo as

Proceedings of the 16th International Workshop on Rapid System Prototyping (RSP'05)
1074-6005/05 $20.00 © 2005 **IEEE**

computed by its IMU unit based on measured time, the MTL for this assertion is:

Rule 2: *(Not explode) Until (IMUdist>=600).*

This assertion has a simulation-time variant described in section 4.2.

3. The torpedo should never navigate above 35 ft. The MTL for this assertion is:

   Rule 3: *Always (depth > 35 ft).*

4. Once the torpedo reaches a depth below 2000 ft, it should disarm within 2 seconds. The torpedo may re-arm once it rises back up to 2000 ft. However, if the torpedo reaches a depth below 2500 ft, it should be permanently disarmed within 2 seconds. MTL for these assertions is:

   Rule 4a: *Always (depth>2000 => Eventually$_{<2\ sec}$ disarmed).*

   Rule 4b: *Always (depth>2500 => Eventually$_{<2\ sec}$ (disarmed && Not Eventually armed)).*

4.2 KTorp Simulation-time Assertions

A simulation-time assertion is an assertion that uses information about the environment not present in run-time. Consider, for example, assertion #2 above. Distance in the torpedo is computed by the IMU unit based on measured time. It is therefore an estimated distance. The simulator on the other hand, contains the actual distance of the torpedo from the mother ship; we denote this distance as A*ctualDist*. Consequently, the following is a simulation assertion variant of assertion #2: "The torpedo should never explode if A*ctualDist*<600 yards", written in MTL as

Rule 5: *(Not explode) Until (ActualDist >=600).*

Note that rule #5 is an example of an assertion about the raw requirements, while rule #2 is an example of an assertion about the design that uses a particular algorithm to estimate distance traveled based on time. Both rules are active during simulation. By choosing a less accurate algorithm for the IMU unit, we have recreated a simulation scenario where rule #5 fails and rule #2 succeeds. Such a scenario identifies an issue that would probably not have been detected during testing using only information available to the torpedo software: *there was a potential for the torpedo exploding while not sufficiently far from the mother ship*. The discovery will cause the system designer to set tighter constraints on the accuracy of the IMU software.

4.3 KTorp Run-time Assertions

Run-time assertions are assertions that are integrated with the run-time control flow of the application. Consider for example, if the torpedo toggles between 2100 ft and 2300 ft for 3 times or more within a 60-second interval then it is prudent to assume that there is something suspicious about the torpedo's behavior. Consequently, the torpedo should be permanently disarmed. The formal assertion (using the time-series constraints of [D2]) is:

Rule 6: *Not Eventually $time$ Repeated$_{>2}$ depth<=2100 && Next (depth>2300 && time'<time+60).*

An action associated with the failure of this assertion will permanently disarm the torpedo. Rule 6 is an example of a requirements assertion that finds its way into the target code. Assume that we implement the system in Java. The exception handling with temporal logic will be done as follows.

(1) Express the assertion using embedded TemporalRover syntax [D3]:

```
/* TRBegin
    TRAssert {Not Eventually $time$ Repeated_>2_ (
        depth<=2100 && Next (depth>2300 &&
                             time'<time+60)) }
    => {$ // empty success action
        throw new MyException("Rule 6 failed");  $$}
 TREnd */
```

(2) In the target Java code at the location requiring run-time checking, wrap the assertion inside a try block and in the catch block do:

```
catch (MyException e) {
    ... recovery action to disarm the torpedo
}
```

(3) Implement the desired exception handling routines in the class MyException.

4.4 Implications of Assertion Classification on Prototyping

Testing is a time- and effort-consuming process. Correct formal specification of system requirements (simulation-time assertions) and assertions describing the correct behavior of control flow (test-time assertions) are essential to test automation. The use of REM, in tandem with rapid prototyping, provides an effective means to validate the correctness of the formal specifications as well as the system design.

Test-time assertions help develop built-in test cases in the design and implementation of the system prototype, and can be embedded in the prototype as probes (code snippets) to support REM. The effectiveness of these lightweight formal methods depends heavily on the designer's ability to understand the requirements and to express them correctly as a formal specification. Prototyping allows designers to valida-

Proceedings of the 16th International Workshop on Rapid System Prototyping (RSP'05)
1074-6005/05 $20.00 © 2005 **IEEE**

tion these test-time assertions early in the development process.

The validation of system behavior requires the development of executable environment models to exercise the system prototype under realistic scenarios. Simulation-time assertions hold the key in deriving the requirements for these environment models. Moreover, the environment may need to undergo constant changes to cope with the evolution of the prototype. Rapid system prototyping allows the construction and modification of these executable environment models rapidly, accurately, and cheaply.

Run-time assertions help increase the robustness of the system software by *armor-plating* them against any unexpected behaviors. For example, we can fortify the software's exception-handling ability via runtime monitoring of temporal assertions, where formal specifications are translated by a code generator into C, C++, or Java statements to be deployed for catching exceptions in the final product during runtime [D1]. The effectiveness of the run-time assertions can only be validated via prototyping, where system designers can inject faults into the system prototype and abnormalities into the operating environment to test the effectiveness of the exception handlers.

5 Runtime Monitoring and Checking of the Assertions

To analyze the safety requirements involving the torpedo in the context of various actors in its environment, we added a *DBRover Connector* module and further refined the *Environment Model* module, resulting in the OMNeT++ simulation model shown in Figure 3, with the User Interface shown in Figure 4.

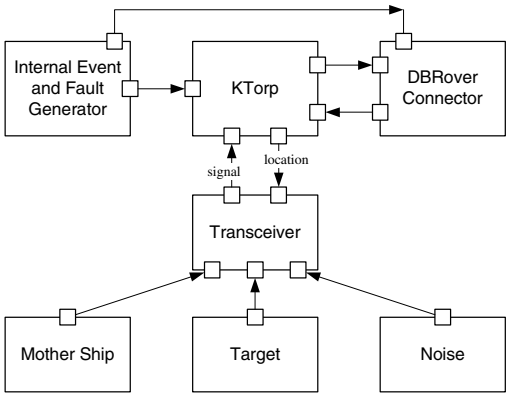

Figure 3. The Phase II KTorp OMNeT++ Model

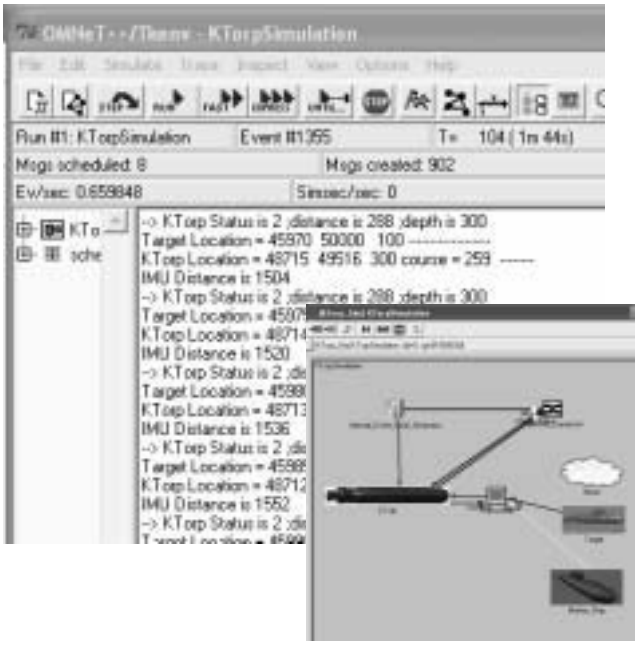

Figure 4. The OMNeT++ User Interface of the Phase II KTorp Simulation

The new environment model allows the system designer to evaluate the correctness of the transceiver algorithms in discerning signals from the three separate modules which model the *Mother Ship*, the T*arget* and the *Noise* of the environment, and the safety requirements involving the locations, headings and speeds of the torpedo, the mother ship and the target.

The *DBRover Connector* module receives the *IMUdist* values from the *Internal Event and Fault Generator* module and the *depth* values as well as the *arm* and *explode* status from the *KTorp* module, executes the code snippets generated by DBRover to evaluate the Boolean conditions of the rule segments and sends the result to the DBRover Runtime Monitor for real-time temporal rule verification (Figure 5). It also uses the feedback from DBRover to permanently disarm the torpedo if Rule 6 is violated.

We also modified the internal code of *Internal Event and Fault Generator* and the *KTorp* modules to provide the option of random fault generation based on the input parameters entered by the system designer at the beginning of a simulation run.

Proceedings of the 16th International Workshop on Rapid System Prototyping (RSP'05)
1074-6005/05 $20.00 © 2005 **IEEE**

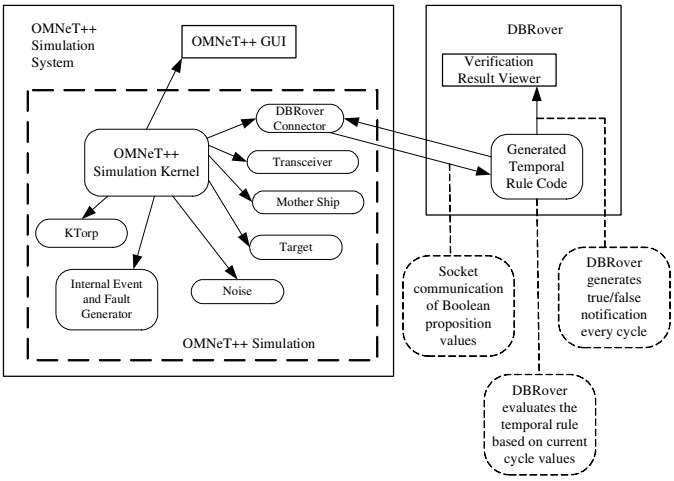

Figure 5. Architecture of the integrated OMNeT++ Simulator / DBRover Run-time Monitor System

6 Conclusions

While formal specification assertions are becoming increasingly popular, they are typically used only in two configurations: (i) for testing, using propositional or temporal assertions, and (ii) in run-time, using propositional assertions (e.g., using the Java assertion feature). In this paper we showed the benefit of identifying temporal assertions for run-time and simulation time purposes. In particular, simulation-time assertions are very useful for the development of the executable environment model, which is quite often neglected in traditional prototyping. Run-time assertions, together with run-time monitoring and exception handling, help increase the robustness of the system under development. Prototyping is essential for testing the effectiveness of the run-time assertions. We also use a simple prototype to show how to instrument a simulation environment to support the evaluation of the temporal assertions as well as the system design.

This paper also highlights the need for new prototype use-cases pertaining to the above-mentioned classification. For example, prototypes used with simulation assertions will often be used to force catastrophic behavior of the kind only available in simulation mode. Similarly, prototypes used with run-time assertions will often be used under illegal contractual circumstances, forcing assertion-based flow-control recovery.

Acknowledgements

The research reported in this article was funded in part by a grant from the U.S. Missile Defense Agency. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotations thereon.

References

[AL] B. M. Atchison and P. Lindsay, "A Safety Validation of Embedded Control Software using Z Animation", *Proc. 5th IEEE International Symposium on High Assurance Systems Engineering*, Albuquerque, NM, Nov. 2000, pp. 228-237

[CDMSS] D. Caffall, T. Cook, D. Drusinsky, B. Michael, M. Shing, and N. Sklavounos, Formal Specification and Run-time Monitoring within the Ballistic Missile Defense Project, submitted for publication.

[CP] 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.

[D1] D. Drusinsky. Specs Can Handle Exceptions. *Embedded Developers Journal*, November 2001, pp. 10-14. (http://i.cmpnet.com/eet/embedsub/2001/nov/Feature1.pdf).

[D2] D. Drusinsky, "Monitoring Temporal Rules Combined with Time Series", *Proc. 2003 Computer Aided Verification Conference (CAV 2003)*, 8-12 July 2003, pp. 114-117.

[D3] D. Drusinsky, "The Temporal Rover and ATG Rover", *Proc. Spin2000 Workshop, Springer Lecture Notes in Computer Science*, 1885, pp. 323-329.

[DS] D. Drusinsky and M. Shing, "Verification of Timing Properties in Rapid System Prototyping", *Proc.14th IEEE International Workshop in Rapid Systems Prototyping*, 9-11 June 2003, pp. 47-53.

[DM] D. Drusinsky, J.B. Michael and M. Shing, "Behavioral Modeling and Run-Time Verification of System-of-Systems Architectural Requirements", *Proc.2nd International Conference on Computing, Communications and Control Technologies (CCCT'04), Vol. VI*, pp. 13-18, Austin, TX, August 14-17, 2004.

[DW] D. Drusinsky and G. Watney - Applying run-time monitoring to the Deep-Impact Fault Protection Engine. In Proc. 28th NASA Goddard Software Engineering Workshop, IEEE (Dec. 2003), pp. 127-133.

[HR] K. Havelund and G. Rosu - Monitoring Programs using Rewriting. In Proc. 16th Annual Int. Conf. Automated Software Engineering, IEEE (San Diego, Calif., Nov. 2001), pp. 135-143.

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

[Pn] A. Pnueli, "The Temporal Logic of Programs", *Proc. 18th IEEE Symp. on Foundations of Computer Science*, 1977, pp. 46-57.

[Va] A. Varga, *OMNeT++ Discrete Simulation System (Version 2.3) User Manual*, Technical University of Budapest, Dept. of Telecommunications (BME-HIT), Hungary, Mar. 2002.