Modular Prototyping of Systems and Environments Using Models Developed with Attributed Event Grammar

Kadir Alpaslan Demir
Department of Computer Science
Naval Postgraduate School
833 Dyer Road, Monterey, CA 93943, USA
kdemir@nps.edu

Abstract

This paper introduces a technique for developing modular prototypes of systems and environment using models developed with attributed event grammar. The motivation underlying the technique is to provide support for rapidly modifying prototypes in order to perform what-if analyses that require updating the models to reflect new sets of system requirements. The modularity is achieved via modeling the necessary system components as independent models. The environment is also modeled and incorporated into the prototype to yield better what-if analyses. We demonstrate the technique using an autonomous homing torpedo system prototype as an example.

Keywords: Rapid prototyping, Modular prototyping, Model-based software engineering, Attributed Event Grammar

1. Introduction

Concept exploration and requirements development is a crucial phase in software development. The cost of fixing an error in the later phases of software development exponentially increases after requirements development [1,2]. Requirements errors are likely to be the most common class of errors and to be the most expensive errors to fix [3]. Better requirements development techniques are still needed. There is no single technique addressing all the problems. Different domains and projects require different techniques.

The main approach for requirements elicitation and analysis includes user and stakeholder interviews [4]. However, some systems operate in unfamiliar environments and do not directly interact with users [5]. Therefore, requirements can not be extracted by interviewing users or stakeholders. Examples of such systems are autonomous systems and analysis and design of these real-time reactive systems pose many challenges [6,7,8]. Autonomous systems accomplish their tasks by reacting to the stimuli in the environment. In this sense, the environment is the real user. This special case necessitates special techniques for requirements extraction. In order to generate the requirements and the resulting specifications for these systems, the developers must be able to predict the system behavior in its environment. The essential way to understand the behavior in the early phases of development is to build prototypes rapidly and run simulations based on them [9].

The importance of rapid system prototyping has been recognized by many researchers [10,11,12] and prototyping reduces cost, program size and programmer effort[10]. Rapid system prototyping helps to identify the requirements and analyze the system behavior based on the system model. However, prototyping just the system is not sufficient for the development of autonomous systems. In order to correctly analyze the system behavior, developers should also model and prototype the environment. Atchison and Lindsay proposed to build models for both the system and the environment for verifying safety requirements [13]. Then two models are exercised in tandem to check whether the system ends up in hazardous states. Our technique also recognizes the need for building two different models and prototypes for the system and its environment.

Traditionally, modeling approaches for software development are primarily focused on the system. Various different technologies are used in modeling system behavior. Examples include UML [14], statecharts [15], timed automata [16], petri nets [17], Temporal Logic [18] and its various extensions, TLCharts [19,6], Z formalism [20] etc. Modeling the system has gained considerable attention. However, even though some of these modeling languages can be used for modeling the environment behavior, for example in [13], modeling and prototyping the environment has not received enough attention. Therefore, this paper focuses on an area that has not discussed in detail. In [13], Z animation is used for modeling the system and its environment (in this case, equipment) for safety validation of embedded control software. Z formal language has an extensive set of mathematical constructs, while attributed event grammar has considerably limited and simpler set of mathematical constructs.

The focus of attributed event grammar is the notion of event. Such a focus is quite useful when modeling environments for the purpose of designing, prototyping, verifying and validating systems that interact with its environment. Because our focus in these modeling activities is the events and corresponding responses rather than the mechanics of how these events are generated. Therefore, just expressing the events and responses in a model with adequate and simple tools is sufficient for our purposes such as concept exploration in the early phases of development. The attributed event grammar is a good candidate for such purpose.

The contribution of this paper is the introduction of using attributed event grammars for modeling both the system and its environment as well as achieving modular prototyping in the early phases of software development. Benefits of modular prototyping for software development is recognized [22,23] but has not received enough attention. In this paper, we address this subject and provide a technique and example for building modular prototypes for the system and its environment.

2. Attributed Event Grammar

Attributed event grammar is an extension of event grammars. Event grammars have been used for program testing, monitoring, and debugging information [26,27,28]. It is possible to generate an event trace as a model of behavior using attributed event grammars. This model of behavior can be used for test automation and system safety assessment [21] as well as prototyping a system, its environment or both.

Attributed event grammar is based on the notion of event. An event is any type of detectable action. It has a beginning, duration and ending. An event also has attributes. For example, timing parameters, variables to store inputs etc. There are two basic relations defined for events: Precedence and inclusion. One event precedes another event if the ending of the event is before than the beginning of another event. An event may include other events. For example opening a file includes checking for permissions, reading the raw binary, converting into readable format etc. These two basic relations are sufficient to represent an event trace for the behavior of a system or an environment. An event grammar specifies the structure of possible event traces. The detailed discussion on attributed event grammar can be found in [21]

3. Overview of the Technique

The technique has two main steps. The first step involves developing the models of the system and the environment using the attributed event grammar. The second step involves creating the prototypes based on the models built in the first step.

3.1. Modeling the system and the environment with attributed event grammar

The first step begins with identifying the components of the system that are likely to be modified during the development due to new information about the environment. In addition, a further refinement may be conducted to identify the components those are likely to be reused in the future for other similar projects. Then, the models of the components are built using attributed event grammar.

The development of autonomous systems without the consideration for its environment will be incomplete and misleading. Therefore, the technique includes environment modeling. The ongoing research about the environment may require modular models. Developers must be able to incorporate new knowledge to the model without affecting other parts of the model. Thus, we build the environment model separately and create a common interface between the system and the environment model. The environment may sometimes necessitate building modular models within the environment modeling. For example, consider the development of a guided missile. The environment for the missile includes weather conditions, noise, and target. All of these generate external events for the missile. The events may or may not interact with each other. The noise level varies depending on the weather and the location. The signals generated by the target may be independent of the location. Creating separate modular models will enable developers to easily integrate newly discovered properties to the environment components.

3.2. Development of Prototypes

The second step is the development of executable modular prototypes. The prototype modules are based on the models built in the first step. It is possible to argue that there is no need for the first step and the prototypes can be built based on the specification. The problem is at the early phases of the development, the requirements aren't clear and the specifications don't exist. The requirements will be clarified based on the results from the execution of the prototype. Only after that, the specifications can be written. Attributed event grammars are formal; therefore, it is possible to analyze the models formally.

We used ONMET++ as the prototyping environment. OMNET++ is an object-oriented discrete event simulator [24]. OMNET++ can be successfully used for simulating various types of systems especially when the concern is events and we used OMNET++ for the simulation of a similar system in [25]. Other prototyping tools can also be used for the second step but there are a couple reasons for the selection of this particular tool. First, OMNET++ is object-oriented which eases the development of modular prototypes. Second, the tool has an inherent structure for timed event traces with a well-designed GUI. Third, the tool is based on C++, a common programming language, with specialized libraries.

4. Example: KTORP

To validate the technique as well as to understand the capabilities and the limitations of the technique, we used the approach on the development of an artificially designed submarine-launched homing torpedo, KTorp. This system is a good example for our purposes in many aspects. First, the environment of the torpedo is underwater. To understand and predict the behavior of this environment from the torpedo's viewpoint is not an easy task. Torpedo homing technologies are driven by underwater acoustics studies. Advancements in the knowledge base of underwater acoustics will help to design better homing technologies. Thus, we need the ability to incorporate newly acquired knowledge in modeling of environments for developing such systems in a cost effective way. Second, KTORP is a complex real-time mission and safety critical weapon system. It is a real world problem driven by real world requirements.

Like many other guided weapon systems, KTorp searches, tracks and destroys its target. The navigational time limit is approximately 35 minutes. The operational depth limit is 35-2000 ft. KTorp has two search modes: Snake and Circular. The torpedo run consists of three basic phases: (i) Enable, (ii) Search, and (iii) Attack or Torpedo Run End. The phases are the same for both search modes but their behavior differs.

The enable phase begins with launching the torpedo. In this phase, the torpedo is disarmed and its sonar is inactive. The goal in this phase is to gain a safe distance from the mother ship (the submarine KTorp is launched from). Otherwise, the torpedo may receive signals from the mother ship and may even attack its own launch platform. The enable phase ends with an internal enable signal indicating that KTorp reached the enable distance (in other words safe distance). If the torpedo control logic does not receive an enable signal, KTorp ends its run just before its battery depletes.

The search phase is automatically initiated at the end of enable phase. The torpedo arms itself and its sonar becomes active. In this phase, KTorp searches for a potential target with a predefined pattern specified with its search mode. In snake search, KTorp's navigation is very similar to a snake's movement. In circular search, KTorp searches its target by circling within a search area. When KTorp detects signals satisfying a certain threshold, the torpedo enters the attack phase. If the torpedo is unable to enter the attack phase due to not acquiring a target, KTorp ends its run with disarming itself.

The attack phase is the final phase before KTorp's final move for the target. In the attack phase, the torpedo still searches for its target but in a focused area with more precision. KTorp decides based on the accuracy of the signals generated by the target. When the signals satisfy a certain threshold, KTorp attacks. In this phase, if the torpedo is unable to detect a target within one minute, it falls back to its search phase.

4.1. KTORP and Its Environment Modeling

The preliminary requirements of KTorp are identified during the concept analysis. These are high level system requirements. However, they are enough for developing the models and exploration of refined requirements. At this phase, the following modular components of the system are identified (Even though, all the modules are modeled using attributed event grammar, in this paper we only present KTorp control logic and environment model):

1. KTorp Control Logic Model: This module is responsible for all the decisions and control actions for the torpedo. KTorp control logic module decides on the phase transitions and the navigation patterns, checks the signals whether they satisfy the threshold or not etc. In order to keep the model understandable and simple, only a certain portion of the system is modeled and presented here. However, even this portion shows that the technique is scalable to real-world applications. In this model, snake search mode is specified while details of the circular search mode are left out. The reason is not to burden the readers unfamiliar with this specific domain. The KTorp model is specified using the event attributed grammar.

```
KTORP MODEL

RULE Torpedo_run
{  int search_mode;
  bool enable_signal_received, depth_limit_reached,
target_acquired, attack_activated, target_attacked,
target_signal_received, snake_attack_start;
  int search_signal_count, search_signal_duration,
attack_signal_count, attack_signal_duration, depth;  }
```

```
Torpedo_run:
 /enable_signal_received=false; attack_activated=false;
   search_mode=1; /* indicates mode is set to snake search */
   target_attacked=false;  depth_limit_reached=false;
   target_acquired=false; search_signal_count=0;
   search_signal_duration=0; attack_signal_count=0;
   attack_signal_duration=0; snake_attack_start=true;
   depth=35; /
(*Enable_phase Torpedo_run_end*) (==2100) (EVERY 1 s);

Enable_phase:
/ENCLOSING Torpedo_run.enable_signal_received =
get_enable_status(); /
WHEN (Torpedo_run.enable_signal_received==false)
   Enable_navigation
ELSE (WHEN (Torpedo_run.search_mode==1)
                    Snake_search ELSE Circular_search);

Circular_search:/*not included to keep the model simple*/;

Enable_navigation: /move_straight();
            ENCLOSING Torpedo_run.depth=get_depth();/
            WHEN (ENCLOSING Torpedo_run.depth>2000)
/ENCLOSING Torpedo_run.depth_limit_reached =true;/ ;

Snake_search:
 WHEN (ENCLOSING Torpedo_run.target_acquired)
Snake_attack_phase ELSE Snake_search_phase ;

Snake_search_phase: Snake_search_move
            /ENCLOSING Torpedo_run.target_signal_received
=get_target_signal() ;/
            WHEN (ENCLOSING
Torpedo_run.target_signal_received == true)
Process_search_signal;

Snake_search_move: /snake_search_pattern_move();
ENCLOSING Torpedo_run.depth=get_depth(); /
            WHEN (ENCLOSING Torpedo_run.depth>2000)
/ENCLOSING Torpedo_run.depth_limit_reached=true;/ ;

Process_search_signal:
/ENCLOSING Torpedo_run.search_signal_count++; /
WHEN (ENCLOSING
Torpedo_run.search_signal_count==1)/start_search_timer();/
WHEN (ENCLOSING
Torpedo_run.search_signal_count==2)
Process_second_search_signal;

Process_second_search_signal:
/ENCLOSING
Torpedo_run.search_signal_duration=get_search_timer(); /
WHEN (ENCLOSING
Torpedo_run.search_signal_duration<6)
        /ENCLOSING Torpedo_run.target_acquired=true);/
ELSE (/ENCLOSING torpedo_run.search_signal_count=1;
start_search_timer(); ENCLOSING
Torpedo_run.target_acquired=false);/ ) ;
```

```
Snake_attack_phase: Snake_attack_move
WHEN (ENCLOSING Torpedo_run.snake_attack_start)
/start_attack_duration_timer(); ENCLOSING
Torpedo_run.snake_attack_start=false; /
ELSE (/ENCLOSING Torpedo_run.attack_duration
=get_attack_duration_timer();/
WHEN (ENCLOSING Torpedo_run.attack_duration<60)
Snake_listen;
ELSE  / ENCLOSING Torpedo_run.target_acquired=false;
       ENCLOSING Torpedo_run.snake_attack_start=true; /);
Snake_attack_move: /snake_attack_pattern_move();
ENCLOSING Torpedo_run.depth=get_depth();/
WHEN (ENCLOSING Torpedo_run.depth>2000)
/ENCLOSING Torpedo_run.depth_limit_reached=true;/ ;

Snake_listen:
/ENCLOSING Torpedo_run.target_signal_received
=get_target_signal() ;/
WHEN (ENCLOSING
Torpedo_run.target_signal_received==true)
Process_attack_signal ;

Process_attack_signal:
/ENCLOSING Torpedo_run.attack_signal_count++; /
WHEN (ENCLOSING
Torpedo_run.attack_signal_count==1) /start_attack_timer();/
WHEN (ENCLOSING
Torpedo_run.search_signal_count>=3)
Process_final_attack_signal ;

Process_final_attack_signal:
            /ENCLOSING
Torpedo_run.attack_signal_duration=get_attack_timer(); /
            WHEN (ENCLOSING
Torpedo_run.attack_signal_duration<=3) Attack;
            ELSE  /ENCLOSING
Torpedo_run.attack_signal_count=1; start_attack_timer();/ ;

Attack:
/ENCLOSING Torpedo_run.target_attacked=true; attack();/ ;

Torpedo_run_end:
WHEN (ENCLOSING Torpedo_run.target_attacked)
/BREAK;/
WHEN (ENCLOSING Torpedo_run.depth_limit_reached)
/disarm_torpedo(); BREAK;/ ;
```

2. Internal Event Fault Generator Model: This module is responsible for generating events internal to the torpedo but external to the KTorp control logic. The reason creating such a module lies in the need of integrating new components along the project life cycle. For example, a new arming mechanism or a new distance measuring unit may be developed throughout the life cycle. In this sense, this type of modularization will help the developers to analyze the system behavior before integrating these new components. It is also possible to reuse these models for further projects. Internal event fault generator consists of various devices and sensors.

3. *Transceiver Model:* This module represents the sonar and transceivers of the torpedo. It is separated from the internal event fault generator module due to its importance in affecting KTorp's behavior. The signals are received and processed by this module. In a real torpedo development, this component will likely to face iterations. Thus, it is modeled as a separate module.

4. *Environment Model:* The environment for KTorp is underwater which is complex and dynamic. The underwater environment varies to due to weather conditions, seasons, and regional differences. This aspect of the underwater environment requires extensive research and reveals the necessity of a modular modeling approach. For the purpose of this research, the environment model is kept simple and abstracted. From the torpedo's point of view, the environment is just a medium for signals. This view is used to construct a common interface to represent different type of signals in the underwater environment. In the model, we assumed that there are two types of signals captured by the transceivers. The first type is generated by a real target (for example a submarine) in the environment. This type of signal is continuous by nature and consists of multiple signals. Simply, if there is a submarine in the area, it will continue to generate signals. The second type of signal is a false echo signal, very similar to the first type. This signal is due to the noise in the environment. False echo signals differ from a real target signal in the number of signals and the frequency they are generated. The model to represent the environment is as follows:

<u>**ENVIRONMENT MODEL**</u>
Underwater_environment:
{ Target_signal_combo , False_echo_signal };

Target_signal_combo:
(* P(0.1) Target_signal *) (==70) (EVERY 30 s) ;

Target_signal: (* DELAY(RAND[0..2]) Signal *) (3..5);

False_echo_signal:
(* P(0.3) Signal *) (==17) (EVERY 120s);
Signal: /generate_signal();/;

4.2. Simulations on Executable Prototype

The prototype based on the models is created using the OMNET++ prototyping environment. For every model specified, a separate executable module is implemented. In addition, main events in the models are implemented with function calls within the prototype modules. This increases the modifiability of the modules. For example, *Enable_Signal* event in the Internal Event Fault Generator model is implemented as a function call in the corresponding executable prototype module.

Enable_signal: P(0.99) / send_enable_signal(); /;

<u>**Implementation of Enable_Signal event in C++**</u>

```
void Internal_Event_Gen:: send_Enable_Signal()
{int enabler_probability = 99;
 int enabler_prob = intuniform(0,100);
if (enabler_prob<=enabler_probability)
{ cMessage *enable_Signal = new cMessage(
"Enable",1);
  enable_Signal->setKind(1);
  send( enable_Signal, "out" );
  ev << "Time= "<< simTime() <<" Enable
Signal is sent.\n";
 }}
```

During this step, another module, system monitor, is added to the prototype to record important events. For the purpose of the study, the module identifies the following events: (i) if the torpedo successfully attacks (ii) if the torpedo is unable to find a target or the torpedo run ends (iii) if the torpedo violates the depth limit. Figure 1 shows OMNET++ conceptual model.

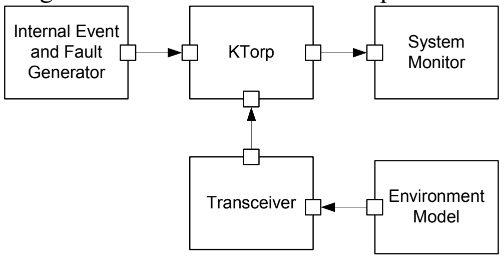
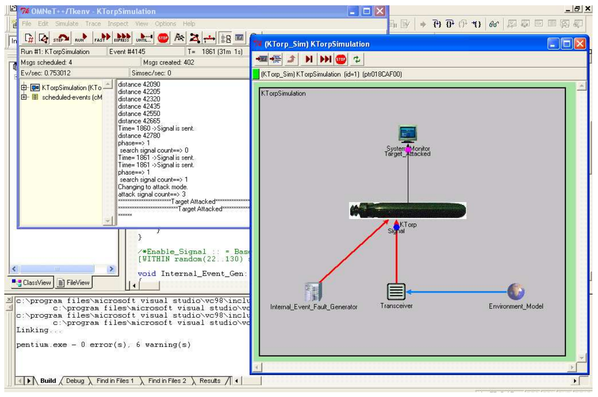


Figure 1. OMNET++ Conceptual World Model

4.3. Simulation Results


OMNeT++/Tkenv - KTorpSimulation
File Edit Simulate Trace Inspect View Options Help
STEP RUN FAST EXPRESS UNTIL STOP
(KTorp_Sim) KTorpSimulation
KTorpSimulation
System Monitor
Target_Attacked
KTorp
Signal
Internal_Event_Fault_Generator
Transceiver
Environment_Model
ClassView FileView
Build Debug Find in Files 1 Find in Files 2 Results

```
Run #1: KTorpSimulation   Event #4145   T= 1861 (31m 1s)
Msgs scheduled: 4   Msgs created: 402
Ev/sec: 0.753012   Simsec/sec: 0
KTorpSimulation (KTo
scheduled-events (cM
(KTorp_Sim) KTorpSimulation (id=1) (ptr018CAF00)
```

```
distance 42090
distance 42205
distance 42320
distance 42435
distance 42550
distance 42665
Time= 1860 ->Signal is sent.
distance 42780
phase==> 1
 search signal count==> 0
Time= 1861 ->Signal is sent.
Time= 1861 ->Signal is sent.
phase==> 1
 search signal count==> 1
Changing to attack mode.
attack signal count==> 3
**********************Target Attacked**************
************************Target Attacked************
******
```

```
}
/*Enable_Signal :: = Base
[WITHIN random(22..130)

void Internal_Event_Gen:
```

```
c:\program files\microsoft visual studio\vc98\inclu
        c:\program files\microsoft visual studio\vc
c:\program files\microsoft visual studio\vc98\inclu
        c:\program files\microsoft visual studio\vc
Linking...

pentium.exe - 0 error(s), 6 warning(s)
```

Figure 2. Screenshot from the Execution of the Prototype

We ran 100 simulations on the prototypes with the default parameters. In the simulations, the prototypes based on the models of the system and the environment behaved as we intended to model. The ability of simulation time monitoring of OMNET++ enables us to understand the dynamics of the interactions between these prototypes. OMNET++ shows the individual signals generated in a visual and textual way. Visual simulations decreases the time required to pinpoint and communicate design problems. Especially communicating the problems in the design to other developers and executive managers is an important issue. Being able to visually simulate such complex behaviors helps the system design process. OMNET++ also enables us to run multiple simulations and log the results in an automated fashion. Achieving these benefits are other important criteria in selecting prototyping and simulation tools.

The result of the simulations is that KTorp was able to attack its target with a success rate of 31%. Figure 2 shows the screenshot from the executions. The success rate may seem low but our interest in this study was not to achieve a particular success rate but to test the technique to assess the behavior of the system and its interactions with the environment based on models. For this particular example, the following issues are identified: *i) Initial requirements are far from satisfaction. ii) Improvements on the environment and system model are necessary. iii) Current simulation parameters may not be realistic.*

Identification of a low success rate in the high-level design of the torpedo shows that the technique is capable of identifying design problems early and in a cost-effective way. The simulation results showed what needs to be done for the future iterations in concept exploration. For example, future iterations should include conducting studies to identify more accurate environment-related parameters, revising the concept of torpedo homing system, revising and improving the initial requirements, conducting other analyses to determine whether to go on with the project or suspend the project until relevant technologies are improved. The study ended at this point since we achieved our research goals.

5. Conclusions and Future Work

This study builds on the work [21] by Auguston et. al. However, there are differences and improvements. First, their work is focused on model-based test automation and testing is conducted after the implementation of the system. Our work focuses on concept exploration and requirements development phases prior to implementation phase. In [21], only environment for the system under test is modeled. In this study, it is extended to modeling both the system and its environment using attributed event grammar. We also emphasize modularity while modeling.

Our experiences in developing the example indicate that the proposed technique is applicable and scalable for similar systems.

This technique is proposed for the concept exploration and high-level requirements analysis phases. The technique is found suitable for these phases due to fact that at these levels the focus is just brainstorming of ideas, concepts and initial requirements. At these levels, developers are not interested in low-level timing constraints, race conditions, detailed efficiency analysis etc. However, the value of validating concepts and high-level requirements in early phases is obvious because fixing incorrect requirements later is too costly. Therefore, we believe the technique adds value to current practice with providing a rapid and applicable framework. The main advantages of the proposed technique may be summarized as follows:

- *The technique provides a mechanism to rapidly validate concepts and initial requirements.*
- *A highly modifiable modular approach is proposed to increase efficiency that can be used for prototyping of software intensive autonomous systems.*
- *Incorporating modular environment modeling into system prototypes will yield better what-if analysis.*
- *The modular executable prototypes will enable scalability for analysis of complex systems.*

The technique provides automation for exercising a large amount of scenarios for better what-if analysis. Currently, we are working on incorporating reliability assessment mechanisms to the technique. After that, the technique will not only provide insights for initial requirements but also will help to assess and predict the reliability of the proposed system.

6. Disclaimer and Acknowledgements

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 any affiliated organization or government.

7. References

[1] B. Boehm, V. R. Basili, Software defect reduction top 10 list, IEEE Computer Jan. (2000) 135-137.

[2] A. Davis, Software Requirements: Objects, Functions, and States, Englewood Cliffs, NJ: Prentice-Hall, 1993.

[3] D. Leffingwell, and D. Widrig, Managing Software Requirements, 2nd Ed., Boston, Addison-Wesley, 2003

[4] S. R. Schach, Object Oriented and Classical Software Engineering,5th Edition, McGraw-Hill, NY, 2002

[5] A. Bredenfeld, and J. Wilberg, Model based multi-level prototyping, Proc. of 10th IEEE Int. Workshop on Rapid System Prototyping, Clearwater, Florida, June 1999, pp. 190-195

[6] K. A. Demir, Analysis of TLCharts for Weapon Systems Software Development, Masters Thesis, Naval Postgraduate School, Monterey, CA, December 2005

[7] K. A. Demir, Meeting non-functional requirements through software architecture: A weapon system example, Proceedings of the 1st Turkish Software Architecture Design Conf., Istanbul, 20-21 Nov. 2006, pp. 148-157.

[8] D. Drusinsky, M. Shing and K. A. Demir, Creation and validation of embedded assertions statecharts, IEEE Distributed Systems Online, vol. 8, no. 5, (2007).

[9] F. Kordon, and P. Estraillier, Complex system prototyping and using environment abstraction, 2nd Int. Workshop on Rapid System Prototyping, Shortening the Path from Specification to Prototype, 1991, pp. 34-46.

[10] L Bernstein, Forward: Importance of software prototyping, Journal of Systems Integration – Special Issue on Computer Aided Prototyping, 6(1), (1996) 9-14.

[11] F. Kordon, and Luqi, An introduction to rapid system prototyping, IEEE Transactions on Software Engineering, Vol. 28, No. 9, September (2002) 817-821.

[12] Luqi, Software evolution through rapid protoyping, IEEE Computer, May (1989) 13-25.

[13] Atchison and P. Lindsay, A safety validation of embedded control software using Z animation, Proc. 5th IEEE Int. Symposium on High Assurance Systems Engineering, Albuquerque, NM, Nov. 2000, pp. 228-237

[14] B. Selic, Using UML for Modeling Complex Real-Time Systems, Lecture Notes in Computer Science, Vol. 1474, Springer Berlin / Heidelberg, (1998) 250-260.

[15] D. Harel, and E. Gery, Executable object modeling with statecharts, Proceedings of 18th International Conference on Software Engineering, IEEE, Berlin, Germany, March 1996, pp. 246-257.

[16] R. Alur, D. L. Dill, A Theory of Timed Automata, Theoretical Computer Science, 126, (1994) 183-235.

[17] J. Desel, and G. Juhas, What is a Petri Net? Informal Answers for the Informed Reader, Lecture Notes in Computer Science, Vol. 2128, Springer Berlin / Heidelberg, (2001) 1-25.

[18] A. Pnueli, Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends, Lecture Notes in Computer Science, Vol. 224, Springer, (1986) 510-584.

[19] D. Drusinsky, Semantics and Runtime Monitoring of TLCharts: Statechart Automata with Temporal Logic Conditioned Transitions, Proc. 4th Runtime Verification Workshop (RV 04), Electronic Notes in Theoretical Computer Science, vol. 113, Elsevier, 2005, pp. 3-21.

[20] J. M. Spivey, The Z Notation: A Reference Manual, 2nd edition, published by J. M. Spivey, Oriel College, Oxford, England, 1998

[21] M. Auguston, J.B. Michael, and M. Shing, Environment behavior models for automation of testing and assessment of system safety, Information and Software Technology, 48 (2006) 971-980.

[22] Luqi, V. Berzins, Rapidly prototyping real-time systems, IEEE Software, Sep. (1988) 25-36.

[23] A. Nketsa, and R. Valette, Rapid and modular prototyping-based Petri nets and distributed simulation for manufacturing systems, Applied Mathematics and Computation, 120, (2001) 265-278

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

[25] D. Drusinsky, M. Shing and K. A. Demir, Test-time, run-time, simulation-time temporal assertions in RSP, Proc. of 16th IEEE Int. Workshop on Rapid System Prototyping, (RSP'05), IEEE CS Press, 2005, pp. 105-110.

[26] M. Auguston, A language for debugging automation, in Chang, S. K., ed., Proc. Sixth Conf. on Software Engineering & Knowledge Engineering, Skokie, Ill., Knowledge System Inc., June 1994, pp. 108-115.

[27] M. Auguston, Lightweight semantics models for program testing and debugging information, Proc. Seventh Monterey Workshop: Modeling Software System Structures in a Fastly Moving Scenario, Santa Margherita Ligure, Italy, June 2000, pp. 23-31.

[28] M. Auguston, C. Jeffrey, and S. Underwood, A framework for automatic debugging, in *Proc. Seventeenth Int. Conf. on Automated Software Engineering,* IEEE (Edinburgh, Scotland, Sept. 2002), pp. 217-222.