International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:1, No:8, 2007

Complexity of Component-based Development of Embedded Systems

M. Zheng, and V. S. Alagar

Abstract**—** The paper discusses complexity of component-based development (CBD) of embedded systems. Although CBD has its merits, it must be augmented with methods to control the complexities that arise due to resource constraints, timeliness, and run-time deployment of components in embedded system development. Software component specification, system-level testing, and run-time reliability measurement are some ways to control the complexity.

Keywords**—** Components, embedded systems, complexity, software development, traffic controller system.

I. Introduction

COMPLEXITY hinders the correct development of software. Some complexity is inherent in the application domain, while additional complexity gets injected by the methods and languages used to implement the system. In this paper we investigate complexities that arise in the development process of embedded systems and suggest some solutions to control it. As a running example we discuss an *autonomous traffic controller* (ATC) system.

Traffic control systems are *safety-critical real-time reactive systems*, characterized by continuous interaction with their environment through stimulus-response behavior. Processes in the system are responsible for synchronization with their environment. The *reactive behavior* refers to two types of synchronization in the system:

- (*stimulus synchronization*): a process reacts to every stimulus from the environment, and
- (*response synchronization*): the system responds in a *timely* fashion so that the environment can make use of the response.

Factors contributing to the complexity of such embedded systems are resource constraints, safety-criticality, concurrency, and the time-dependent functionality required of the artifacts they control. The complexity of system requirements permeates into several stages of development, and may lead to faulty designs and unpredictable failures. This type of complexity should be resolved at early stages. As remarked by Burbeck [6], *"once complexity has gotten out of control, it takes control"*.

In an earlier paper [2] we viewed the complexity of real-time reactive system in five dimensions: *representational*, *architectural*, *internal*, *functional*, and *computational*. We may regard the computational complexity to include the complexity arising from resource limitations and the architectural complexity to include complexity arising from adaptation requirements. Representational complexity considers the trade offs between different choices in the notation for an unambiguous representations of system model, system interaction, and system behavior. Cognitive complexity is part of representational complexity, which in turn is related to the mode of human-system interaction in different levels of abstractions. Architectural complexity is viewed in terms of how the software components interact through message passing mechanism, without considering the individual complexity of the components. Internal complexity is the complexity of individual software component. The functional complexity characterizes the dynamic performance of the system. Computational complexity quantifies the time and resources required to complete the process and is covered by the study of scheduling, resource allocation, and algorithmic efficiency.

In the requirements analysis and specification phase, the focus is on containing the representational complexity of graphical and formal presentations for capturing and conveying information on environmental and system objects. Architectural and internal complexities in embedded systems originate in the problem domain in the form of functional requirements and stimulus-to-response timing constraints and characterize the design product. The architectural complexity of the design results from the interactions among components, which may be distributed and interact through message sequences described in the architectural specification of the system. The internal complexity of its objects originates from their internal behavior. In the verification and validation phases the computational complexity of the process algorithms is considered. The complexity in testing phase is a function of the architectural complexity related to the testability of the software components, and the computational complexity of testing process algorithms. The functional complexity is defined as the amount of work performed by an application in a predefined time slice.

In the following sections we discuss methods to augment CBD in order to control the complexity in embedded system development.

II. Traffic Controller Problem - case study

We consider an autonomous traffic control system as one which adapts itself to the traffic patterns and regulates the traffic in a safe manner. The traffic controller model developed in [3] assumes that at the proximity of the intersection, each road is divided into six lanes; there are three lanes for *incoming* traffic in each of the northbound, southbound, eastbound, and

Department of Computer Science, University of Wisconsin-LaCrosse La Crosse, WI, 54601, USA. Email: {zheng.mao@uwlax.edu}
Department of Computer Science, Concordia University, Montreal, Quebec H3G 1M8, Canada. E-mail: {alagar@cse.concordia.ca,vasualagar@ios.ac.cn}

westbound directions. In every direction, vehicles in the right lane turn right, vehicles in the middle lane go straight, and vehicles in the left lane turn left. Vehicles approaching in a lane enter the crossing on a first-in-first-out basis. Vehicles cross the intersection in a finite amount of time; no vehicle stops in the intersection. Incoming vehicles in the four *right* lanes are allowed inside the crossing independent of any other lane; they are collision-free. However, traffic lights regulate all lanes. Incoming vehicles in the *middle* and *left* lanes need to wait until they are granted permission to enter the crossing. Vehicles in at most two middle or left lanes can be granted access simultaneously, only if the two lanes are collision-free. When an approaching vehicle is detected in a middle or a left lane ml_1, the controller for that lane requests for access rights to the crossing from the arbiter. If the intersection is empty, or only right lane vehicles are inside, then vehicles in lane ml_1 gain access immediately. If the intersection is already opened to one middle or left lane ml_2, then vehicles in lane ml_1 gain access only if lanes ml_1 and ml_2 are collision-free. If the intersection is already opened to two middle or left lanes ml_2 and ml_3, then vehicles in lane ml_1 need to wait until one of the lanes relinquishes access rights to the crossing. If lane ml_2 returns access rights, then lane ml_1 must be compatible with lane ml_3 in order to be granted access. Otherwise, vehicles in lane ml_1 must wait for lane ml_3 to give up its access rights. When a lane requests for access rights, the lane queues up for allocation. Access is granted to lanes in the order in which they request for permission, subject to collision-free property. That is, a vehicle v_2 that arrives at the intersection later than another vehicle v_1 may be given access to the intersection if the vehicle v_2 satisfies the collision-free property, whereas the vehicle v_1 has failed to satisfy the property. When a lane obtains access rights, it must surrender these rights within a certain time interval.

This problem has the following complexity characteristics:

1) The environment that the system is to monitor and control is complex, in the sense that no precise model of it exists. An approximation would be to consider probability distribution for the arrival pattern of vehicles at the intersection. In reality traffic is sporadic, heavy in certain directions during certain periods of time. As such a general probabilistic model cannot be given.
2) The number of vehicles that compete to cross the intersection during a given interval of time cannot be clearly comprehended. The number of traffic patterns grows exponentially and hence no efficient (optimal) algorithm to regulate them with minimal delay can possibly exist.
3) The traffic pattern is *non-linear* and evolves over time. Hence, the topology of the system and the interaction of system components are subject to change. This adaptive complexity impacts on the architecture of the system, thus raising the level of architectural complexity.
4) The resources that are necessary for the proper functioning of the controller must be adequately sufficient to handle the internal book keeping and system computations. Resource complexity arises because of the need to monitor and regulate the dynamically changing traffic pattern.
5) Functional and computational complexities affect the performance of the system, in particular both *liveness* and *safety*. The system has the liveness property if no vehicle waits indefinitely at the intersection before being allowed by the controller to cross the intersection. The system has the safety property if no two vehicles collide while crossing the intersection.

III. Solutions for Controlling Complexity

In the last few years CBD has received much attention from software developers. From the theory side, component models and their representations have emerged [1], [9]. From the practical side, necessary technologies have emerged for supporting CBD of large systems. However, CBD for embedded real-time systems differs from general CBD for traditional applications: *the components required for an embedded system are loaded at run-time and bindings between them are created by a middleware*. Thus it is most crucial to deal with adaptation complexity and resource complexity. Yet CBD has the potential to control the complexity arising in embedded software development provided techniques for controlling the complexity is added to the basics of component technology.

Component models developed by Sun Microsystems are JavaBeans[1] and Enterprise Javabeans[2], and the model developed by Microsoft Corporation is COM[3]. These component models differ in many technical details, while sharing some important properties:

- Components are reusable, replaceable parts of a system.
- Special components, called *commercial-off-the-shelf* (COTS) can be bought and fitted with other components developed in-house. This further increases the complexity of system analysis.
- Components may be distributed over several computers.
- The interfaces to a component indicate the services *required* and *provided* by it. Such services may affect the internal state of the component.

A. Defining Components

Representational and cognitive complexities can be met by choosing a good notation to describe components. Visual representations help to understand the non-linearity in component interactions. Formal textual notations must accompany visual notation.

A component type $\mathbf{T} = \langle \mathbf{F}, \mathbf{A} \rangle$ defines a black-box view, called frame $\mathbf{F}$, and a grey-box view, called architecture $\mathbf{A}$. The type $\mathbf{T}$ may include more than one grey-box view. The black-box defines the interface types, and the particular grey-box view $\mathbf{A}$ as a structured implemented version of $\mathbf{F}$. An interface of a component is an instance of either *notifies-interface* type or *receives-interface* type. The architecture is primitive if its structuring is to be provided in an

[1] http:\\www.javasoft.com/beans/
[2] http:\\www.sun.com./products/ejb/
[3] http:\\www.microsoft.com/com

underlying implementation (outside the scope of component specification language). A non-primitive architecture includes several subcomponents *nested* to several levels. A specific implementation of a non-primitive structure is obtained by (i) instantiating adjacent level subcomponents, and (ii) specifying the interconnection between subcomponents by means of their interface ties. The four kinds of ties between the interfaces of two component instances A and B are as follows:

- *binding*($\xrightarrow{bind}$) of a receives-interface to a notifies-interface between two components at the same level of nesting (assume that all components are subcomponents of the system),
- *delegating* ($\xrightarrow{delg}$) from a receives-interface of a component A to a receives-interface of a subcomponent B of A on the adjacent level,
- *subsuming* ($\xrightarrow{subs}$) from a notifies-interface of a component B to a notifies-interface of a component A, where B is a subcomponent of A on its adjacent level, and
- *exempting* ($\xrightarrow{exem}$) an interface of a component from participating in the architectural connection.

Following the above conventions we can describe the architecture of an component-based embedded system. We encapsulate each constituent in the system as a component and describe their bindings. For example, in the case of the traffic-controller system, the component **ControllerL** models the system that controls the traffic in the left lane, component **ControllerR** models the system that controls the traffic in the right lane, and **ControllerM** models the system that controls the traffic in the middle lane. Vehicles and traffic lights within a lane are monitored and regulated by the controller for that lane. Thus, the environment for a controller, say **ControllerL**, is the component **EnviornmentL** composed from **Vehicle** components, and **Light** components in the left lane. Since, the number of vehicles in a lane dynamically changes, the environment components **EnviornmentL**, **EnviornmentM**, and **EnviornmentR** model dynamically changing subsystems. The environment of the **Arbiter** component is the component composed from the three environment components. We may assume that a main component called **TrafficSystem** exists which interacts with **Arbiter** component for providing system-level resources and computing facility.

A visual model bridges the communication gap between the different software development teams. Borrowing UML notation we show the component hierarchy for left/middle lanes in Figure 1. The figure is only a partial view of the ATC system. In the figure we use the notation $\langle\langle Porttype \rangle\rangle$ to combine the *notifies-interface* type or *receives-interface* type. An event decorated with ? is an input event, and hence is received by a component at its *receives-interface*. An event decorated with ! is an output event, and hence occurs at the *notifies-interface* of a component.

B. Specifying Components -External View

Visual diagrams are to be accompanied by formal descriptions. A formally described visual model helps to precisely interpret the details shown in the picture during subsequent development stages. The description includes the external and

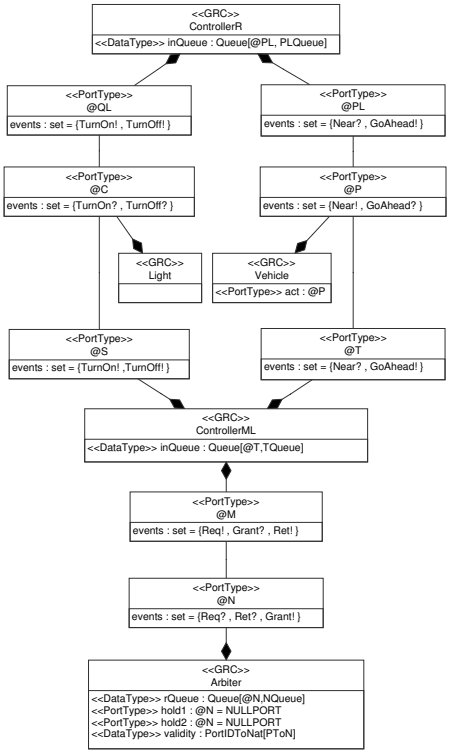

Fig. 1 Class Diagram for Road Traffic Control System

internal properties of the component described. We briefly explain the specification approach that describes the external view of components. Such a precise description reduces the effect of architecture complexity.

A component is specified both as a black-box and a gray-box. The black-box specification gives the specifications of the component interfaces. The gray-box view gives the interaction between a component and its immediate sub-components. Notice that, the sub-components of a sub-component do not enter into the gray-box specification of a component. For a primitive component, namely the component that does not have a sub-component, only black-box specification is given. As an example, a **Vehicle** component is primitive.

a) **Vehicle Interfaces***:* There are two interface types:

Interface Specification	Description
interface **vehnotML** { *void Approaching(out string vid)* }	The vehicle notifies the controller.
interface {**VehrecML**{ *void GetToken(in string did)* } }	The vehicle receives the go ahead to cross.

b) **Autonomous Traffic System** ATS*:* It is composed of the structures *TrafficSystem*(TC) and *Arbiter*AC. The *notifies-interface* of an instance of one component is linked to the *receives-interface* of an instance of the other component. It is a closed system and hence has no external interface.

Frame *Traffi c System ATS*{ };	**architecture** *Traffi c System ATS* { **inst** *Traffi cSystem* TC, *Arbiter* AC; **bind** $TC : TCCAr$ **to** $AC : ACCCn$, $AC : ACCCr$ **to** $TC : TCCAn$; };

The architecture protocol of a visual structure is a behavior protocol that describes the gray-box behavior of the structure. Recall that the gray-box behavior is based only on the direct subcomponents in the structure. It describes the interplay between the components as a trace on the invocations of the methods in their interface specifi cations. The acceptable order of method invocations on an interface defi nes the protocol of that interface. There is only one method at the interface specifi cation given above. Hence the protocol at each interface is the method invocation. Frame protocol specifi es the acceptable interplay of methods at the interfaces in the frame. Component specifi cation help the developer reuse components. Without such a specifi cation reuse efforts will not fully benefi t the CBD process.

C. Specifying Behavior - internal

A component's behavior must be specifi ed as an *Extended State Machine*(ESM). Such a description is fundamental to analyzing the system and develop an implementation. The state transitions in an ESM are specifi ed in *guard-action* notation. The guard typically includes time constraints for the action implied by it. The action may be given declaratively as well as augmented with Java code for automatic code generation, and generating test oracles. COTS are usually delivered without a source code. In order to integrate them with other components both a black-box specifi cation (interfaces) and its behavior specifi cation (internal) should be provided.

A middleware is necessary for implementing a component-based system. The middleware Enterprise JavaBeans[4] defi nes a component architecture for building distributed, object-oriented applications. Since a complete specifi cation of the middleware is hard to obtain, a subset of the features such as resource pooling, and remote invocation, may be modeled as an ESM. The full behavior of the system is the combined behavior of the ESMs of the components and the ESM of the middleware.

Let us consider an application A and a system S that solves A. The system S can be constructed as a collection of components $\{C_1, \ldots C_n\}$ that interact among themselves in solving the problem A. Each component's *state* of S is described by the set of static attributes and their current values. Since a component's total behavior is the cumulative effects of its state changes (internal behavior) and message passing (request from another component to perform a service) (external behavior) the total structural complexity of the software system is a function of the *internal complexity* (of the object's internal behavior) and the *architectural complexity* (of the interactions between the objects). By combining the behavior models and the architectural model we should be able to assess the *reliability* of the system.

Probabilistic reliability prediction [9], and measuring reliability from stochastic system model [8] are two signifi cant approaches that should be added on to CBD of embedded systems. In [8] a Markov model and methods are given to compute the reliability prediction as the system evolves. The reliability of the system is calculated periodically, as defi ned by the system administrator. A component whose reliability falls below an acceptable threshold value should be replaced by a behaviorally equivalent component. Based upon the formal descriptions of behavior given above, it should be possible to check the behavior equivalence of two components. A component to replace an existing component is chosen if and only if their interface specifi cations and behavior specifi cations match. Moreover, the chosen component should have been tested both for its internal and external behavioral properties. Thus, early prediction, based on the architecture and behavior models, help to reduce the development time and improve software quality. The question is, how to test components. We address this issue in the section IV.

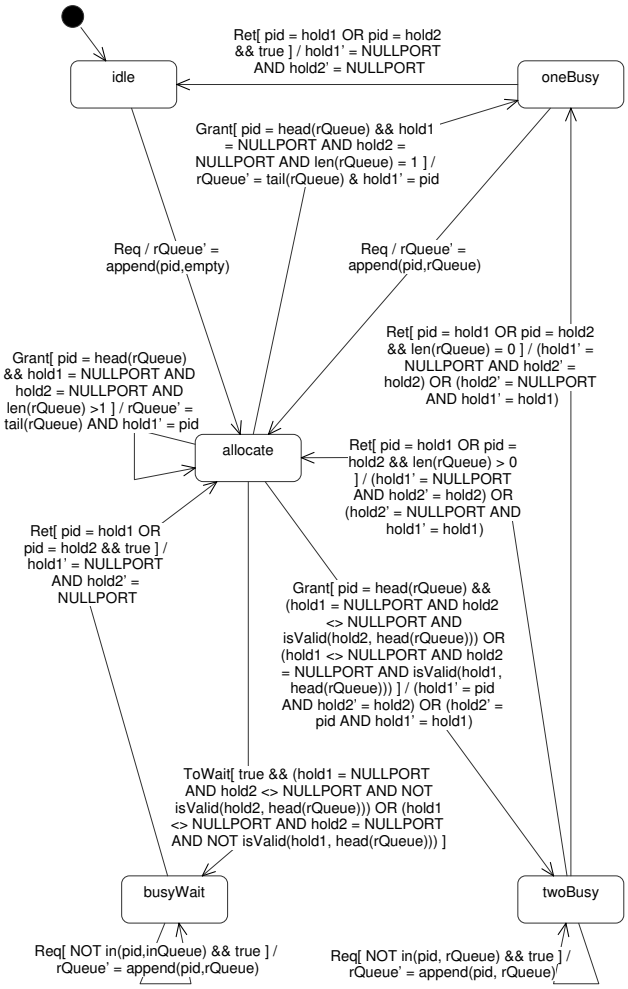

Fig. 2 ESM for Arbiter

[4]http: \\java.sun.com/products/ejb

D. Traffic Controller Example Revisted

We briefly discuss the behavior of **Arbiter** component. The behavior of **Arbiter** component is shown in the ESM of Figure 2. It has a single thread of control; events can not occur concurrently. However, concurrent communication is possible in the overall system that includes the arbiter object . The arbiter may allocate the resource to two controllers concurrently if the lanes they control are collision-free. There are twelve patterns of traffi c flow involving concurrent crossing of the intersection by vehicles in two middle or left lanes without collision. The three resource allocation scenarios of the Arbiter are as follows:

1) In state *allocate* only one controller is accessing the resource, and there are pending requests in the queue.
2) In state *oneBusy* only one controller is accessing the resource, and there is no pending request.
3) In state *twoBusy* two controllers, one for middle lane and one left lane utilizing the resource are compatible; that is vehicles monitored by these controllers do not collide.

In each case, collision inside the intersection is precluded. This can be proved by mutual exclusion property on the guards of the corresponding transition specifi cations. When the arbiter has allocated the resource to one controller and assessed that the next controller in the queue is not compatible with the one already holding the resource, it goes into the state *busyWait*. The resource is not allocated to the controller at the front of the queue. Thus, when the arbiter is in state *busyWait*, vehicles in only one middle or left lane will be allowed to cross the intersection. Informal analysis is helpful, but will often fail to reveal deadlock or starvation in the system. Often even informal analysis is diffi cult to conduct. As an example, for a simple system shown in Figure 3 there are thousands of states and it is impossible to exhaustively analyze by inspection the safety property in all states. It is essential to investigate formal verifi cation and validation methods. In [7] a simulated validation approach and a tool based on it are given.

IV. Testing Component-based Systems

Testing and debugging embedded systems are as diffi cult as formal verifi cation. Some of the diffi culties are:

- For COTS white-box testing cannot be conducted because the source code is not supplied with the component. A black-box testing of COTS is possible provided a specifi cation of its interfaces and protocols are supplied by the manufacturer. When only informal description of COTS is given the testing process is error-prone.
- Middleware is an integral part of the component-based software. It is not possible to precisely characterize all the middleware functionalities. Yet, testing of the middleware coupled to the components and their bindings must be done.
- Black-box testing of components should test the interfaces, and the behavior. Unless a formal description of the component interfaces and behavior are known black-box testing is not possible.

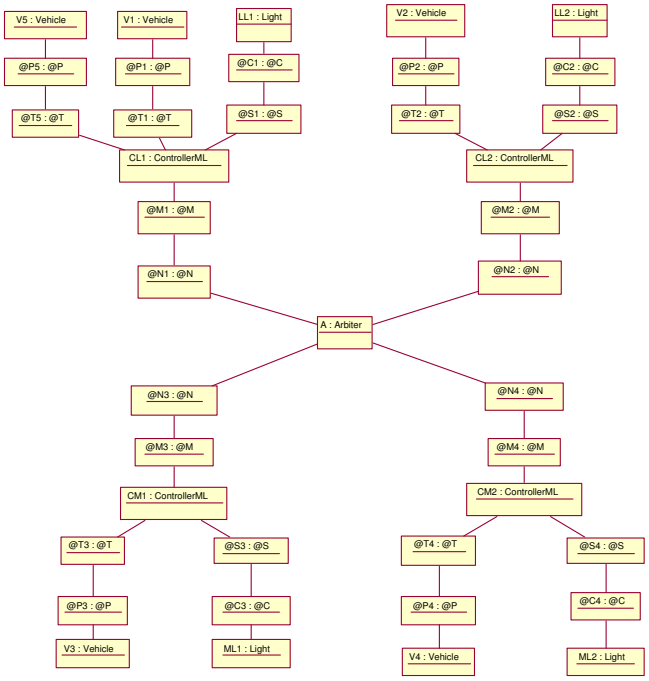

Fig. 3 Multivehicles System

- Component producers are responsible to test the source code of components. Users of components may not be able to conduct white-box testing even when the source code is available to them. There are two reasons for this: (1) the internal structures are not fully observable in source code, and (2) components are assembled at run-time for embedded systems, and real-time white-box testing is not feasible.

Based on the above observations we may draw the following conclusions:

1) Component manufacturers should certify their products.
2) COTS must be accompanied by their formal descriptions.
3) Testing must be a parallel activity to system development.
4) A repository of tested components, and a retrieval system based on component interface descriptions must be provided.
5) System descriptions must be symbolically simulated to assess the reliability of tested components for use in different application domains.

We contend that the notations that we have used for internal and external descriptions of components are conducive for generating test cases for black-box testing. The methods that we have developed earlier [11], [4], [5] are applicable here, the only extension required is modeling the interaction between components and the middleware.

V. Conclusion

In this paper we identified different complexity types that must be dealt in the development of embedded systems. We offerred solutions to deal with them. In our research on real-time reactive system development we have been developing a number of tools to assist the development and analysis phases of the system. We have already applied the testing and reliability prediction tools for the ATC. The design of these tools is both flexible and extendable so as to be applied to other examples of component-based embedded systems.

Acknowledgement

This work is partially supported by a grant from the Natural Sciences and Engineering Research Council (NSERC) of Canada. Part of the work was completed when the second author was a Visiting Professor at the Institute of Software, Chinese Academy of Sciences, Beijing, China.

References

[1] M. Akerholm, A. Moller, H. Hansson, and M. Nolin. *Towards a Dependable Component Technology for Embedded System Applications*. In Proceedings of the 10^th IEEE International Workshop on Object-oriented Real-Time Dependable Systems, Sedona, Arizona, U.S.A., February 2005.

[2] V.S. Alagar, O. Ormandjieva, M. Zheng. *Managing Complexity in Real-Time Reactive Systems*. In Proceedings of ICESSC 2000.

[3] V.S. Alagar, D. Muthiayen. *A Rigorous Approach to Modeling Autonomous Traffic Control Systems*. ISADS2003, Pisa, Italy, pp.193-202.

[4] V.S. Alagar, O. Ormandjieva, M. Zheng. *Incremental Testing for Self-Evolving Systems*. Proceedings of International Conference on Quality of Software, Dallas, U.S.A., November 6-7, 2003.

[5] V.S. Alagar, O. Ormandjieva, M. Chen, M. Zheng. *Automated Test Generation from Object-Oriented Specifications of Real-Time Reactive Systems*. Proceedings of 10th Asia Pacific Software Engineering Conference, Chiang Mai, Thailand, December 10-12, 2003.

[6] S. Burbeck. *Real-time complexity metrics for smalltalk methods*. IBM Systems Journal, June 1996, pp. 1-28.

[7] S.H. Liu. *Simulated Validation of Real-Time Reactive Systems with Parametrized Events*. Master of Computer Science Thesis, Concordia University, Montreal, Canada, October 2003.

[8] Olga Ormandjieva. *Quality Measurement for Real-Time Reactive Systems*. Ph.D thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, January 2002.

[9] R.H. Reussner, I.H. Poernomo, and H.W. Schmidt. *Reasoning about Software Architectures with Contractually Specified Components*. Component-Based Software Quality. LNCS 2693, Springer-Verlag, pp. 287-325, 2003.

[10] H.W. Schmidt. *Trustworthy Components:compositionality and prediction*. Journal of Systems and Software, Elsevier Science Inc, 65(3):215-225, 2003.

[11] M. Zheng. *Automated Test Generation From Formal Specification of Real-Time Reactive Systems*, Ph.D. Thesis, Department of Computer Science, Concordia University,Montreal, Canada, 2002.