Compositional Model-Driven Design of Embedded Code for Energy-Efficient Buildings

A. Mady, M. Boubekeur and G. Provan
CCSL, Computer Science Department, University College Cork,
Lancaster Hall, 6 Little Hanover street,
Cork, Ireland.
{mae1, m.boubekeur, g.provan}@cs.ucc.ie

***Abstract*- In embedded software development, model-driven design is well recognized. In this article we describe a compositional model-driven approach for auto-generating embedded code for improving the energy efficiency of building automation applications. We show how we can use a component-based hybrid-systems modelling framework to generate models for simulation and verification. Then, we auto-generate embeddable code from these models. We empirically demonstrate this approach using the hybrid-systems tool Charon, for the domain of energy-efficient lighting control for smart buildings. The paper provides a detailed description of the code generation steps and outlined some results obtained through a simple lighting example.**

Keywords- Model-Driven Design, optimized building operations, hybrid systems, IFC, Embedded Code-Generation, lighting system.

I. INTRODUCTION

Building models using a compositional model-driven approach is becoming more critical, given the increased scale of systems that are being modelled. For example, in the domain of smart automation of large buildings, such as multi-storey office towers or chip fabrication plants, it is infeasible to build models by hand for such large systems. In such cases, using a component-based tool can significantly improve the speed and reduce the cost of modelling and verification.

Of equal importance to adopting a model-driven approach is the need to make use of industry standards and existing component libraries, since generating a component library from scratch is also expensive and time-consuming. In this article we make use of the IFC (Industry Foundation Classes) standard for the construction industry. IFC is an open standard for CAD (Computer-Aided Drafting) programmes, as used by engineering, construction and architecture companies. Further, IFC is used as a basis for Building Information Modelling (BIM), which is our primary application domain. By adopting IFC, we can make use of standard BIM component libraries, e.g., [1].

In this article we assume that building automation models must be represented using hybrid systems models [2], since one needs to represent both discrete-valued and continuous differential-equation-based relations. We show how we can use component-based hybrid systems models to generate systems models, and then auto-generate embeddable code for a distributed sensor/actuator network.

We demonstrate our approach on a lighting model for a building, in which lights are used to both minimize energy usage and enhance occupant comfort across a range of occupants. Our framework allows users to express preferences for interior lighting levels, and the control system accommodates such preferences over all occupants within a zone. Give a preferred light level; the control system minimizes energy use for the lighting system by controlling blinds for exterior lighting and turning lights on only when a zone is occupied.

Our contributions are as follows. First, we demonstrate how a component-based, hybrid-systems framework can generate verifiable models for simulation. Second, we show how this process can auto-generate embeddable code from these models. Third, we empirically demonstrate code generation steps using the hybrid-systems tool Charon [3], within the domain of energy-efficient lighting control for smart buildings.

The reminder of paper is organized as fellows: Section 2 presents our compositional modelling methodology. Section 3 describes the hybrid modelling and the use of Charon. It also discusses the incorporation of formal verification techniques. Section 4 illustrates the code generation rules from Charon to embedded Java. Section 5 illustrates our approach through a simple lighting system. Section 6 provides a discussion of our work and outlines future perspectives.

II. COMPOSITIONAL MODELLING PROCESS

A. Modelling Process

Engineering design has adopted the principles of modularity, regularity and hierarchy as a key to cost-effective and reliable design, both in theory and practice [4]. As the complexity of technological systems increases, module re-use increases, based on (a) symmetrical and regular structures and (b) developing standards for components and dimensions, since this regularity and component-based methodology translates into reduced design, fabrication and operation costs.

There are currently several modelling tools for building models for smart building applications, such as MATLAB/SIMULINK [5], Dymola [6], and Charon [3].

We first give a brief overview of the process of creating models from a component library within a Model-Based framework. We assume that we can create/redesign a system-

level model by composing components from a component library [7, 8]. We call a well-defined model fragment a *component*. We assume that each component can operate in a set of behavior-modes, where a mode *M* denotes the state in which the component is operating. For example, a pump component can take on modes nominal, high-output, blocked, and activating.

We define two classes of components: primitive and composite. A *primitive component* is the simplest model fragment to be defined. For such a component we specify the inputs *I*, outputs *O*, and functional transformationφ, such that we have O = φ(*I*). In this article we will assume that we specify our systems as hybrid systems [2], in which case our primitive component transformation functions will have hybrid systems semantics.

A *composite component* consists of a collection of primitive components which are merged according to a set ξ of composition rules [7]. In this article we assume standard composition rules; specifying the semantics of composition is beyond the scope of this article, and we refer the reader to [7, 8] for details.

A set of (primitive/composite) components defines a *component library*. In this article we assume a component library consisting of sensors, actuators, human-agent models, and building components such as lights, windows, rooms, etc.

B. Using IFC Classes for Ontology Development

In order to facilitate component integration and code generation throughout the different implementation phases (in particular the design of common interfacing platform), we will be using IFC [9] representations for both building geometry and services.

IFC defines an open international standard that describes an exchange format for information related to a building and its surroundings. It captures within an intelligent BIM both building geometry and components, and also defines their interactions. The current IFC standard version 2x3g defines all information with a globally unique ID using an internationally-agreed ontology. Therefore, IFC offers a great facility for sharing relevant building information across heterogeneous services and applications.

In the same spirit of the IDM (Information Delivery Manual) specification [9] we intend to define/extend a communication dialect that includes interfaces definitions at the high level component as well as the network layer; in particular, we focus on interfaces with the sensors and actuators. Our aim is to specify exactly what information is to be exchanged in each defined use-case scenario. For example, when considering a lighting controller for a particular zone, the information about the size of that zone, the window dimensions and the position of the lights can provide data to perform detailed tuning of the simulation results.

III. HYBRID SYSTEMS MODELLING USING CHARON

Charon is a high-level language for modular specification of multiple, interacting hybrid systems, and was developed at the University of Pennsylvania [3]. The toolkit distributed with Charon is entirely written in Java, and provides many features, including: a GUI (Graphical User Interface), a visual input language, an embedded type-checker, and a complete simulator. Through an intermediate XML format the graphical editor converts the specified model into Charon source code. The Charon toolkit can interact with and/or use other external Java programs. The simulator itself is an executable Java program.

Charon adopts a hierarchal modelling framework based on the statechart modelling technique. A hybrid system is described in Charon as follows [10]:

- *Architectural hierarchy*: The architecture of systems is described with communicating agents. Those agents share information through shared variables or communication channels. Agents are either atomic or composite.
- *Behavioural hierarchy*: A mode is a construct for the hierarchical description of the behaviour; it has well-defined control, entry and exit points. Transitions between modes are enabled when a condition called guard becomes true. Charon provides differential and algebraic constraints representing continuous dynamics and invariants forcing a continuous flow to satisfy a condition. The language also supports the instantiation of a mode for the reuse of mode definitions.
- *Charon variables*: Charon provides two types of variables, continuous (analog) and discrete. Analog variables are updated continuously while time is flowing. Conversely, discrete variables are modified instantaneously only when the modes of an agent change. The values of discrete variables do not change in a time flow.

A. Modelling

As stated earlier, Charon offers features that correspond to our multi-level modelling approach ideas, in particular hierarchy and modularity. We start at an abstract level by defining an abstraction of the different subsystems involved in a building. In a refined multi-level approach, the sub-systems are defined with successively more detail until we reach the implementation level. At different stages, simulation, diagnosis, verification and implementation are performed using the appropriate models.

A top-level model can be a set of agents representing the different services within a building. Other agents can be added at this level to model the building topology in a form that is useful for the overall system. The agents that model the environment appear at different levels of the modelling hierarchy depending on the modelling needs. We notice here that when we consider isolated particular sub-systems, an abstraction of other sub-systems constitutes part of the environment.

Fig. 1 shows a modelling hierarchy for a simple lighting system, showing a subset of the hierarchical agents and modes used to model the system. In this lighting system, a controller aims to maintain the interior light at a set-point when people are present, by setting window blinds to control the exterior light levels, and by turning on/off interior lights. It starts from a high-level representation of the model using

agents (Blinding, Person and Light), and ends using a final state machine describing the behaviour at the modes level.

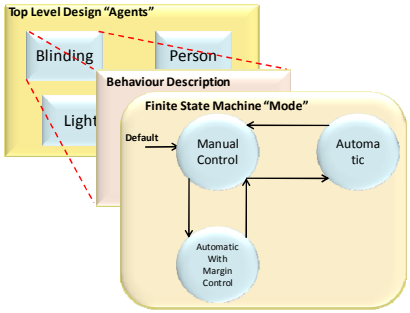

Figure 1: Hierarchy Modelling in Charon

B. *Formal Verification*

When considering building automation controls, the verification process consists, in the best case, of checking systems separately through a simulation process. Our objective is to augment the simulation with formal verification techniques which will formally verify and improve the integration of such systems. This will certainly contribute to the reliability and robustness of the overall system. However, due to the complexity of the new building systems in terms of size and nature, e.g. continuous behaviour, optimization techniques need to be considered. In our work we adapt two widely-recognized and fundamental methods to counter this phenomenon: abstraction and compositional reasoning.

Abstraction is used, for example, to transform hybrid systems to discrete systems in order to access standard model checking tools, e.g. UPPAAL[11]. In cases where continuous-valued parameters are essential to the property to be verified, compositional techniques are considered. The use of Assume-Guarantee techniques, and of compositional reasoning where the specification of a subsystem is only fulfilled under assumptions about the rest of the system, helps to handle the notorious complexity when analysing hybrid systems [12].

In this section we illustrate the simplest use of formal methods within our platform. At the modelling level, a verification option is available when performing the simulation step. Indeed, Charon offers the "assertion" feature that allows to test during the simulation if certain property is respected; otherwise it quit the simulation and displays a failure message.

For example, the following assertion ensures that the internal light is always between 300 and 400 Lux when a user is present in the controlled zone.

```
assert{internal-light >= 300 and  internal-light <= 500}
```

For each simulation instance, Charon checks the correctness of the corresponding boolean expression, proving the absence of any errors.

Since the current version of Charon doesn't provide a fully-featured platform to perform model checking, we have chosen a tool dedicated to the verification of hybrid systems called Phaver [12]. Phaver is a new tool for verifying safety properties of linear hybrid automata, and provides infinite-precision arithmetic in a robust implementation, on-the-fly over-approximation of affine dynamics, and supports compositional and assume/guarantee-reasoning.

The system initially modelled in Charon is totally or partially re-written in Phaver, since the two models are similar except for the hierarchy, which is more evolved in Charon. Model transformation is thus a straightforward task given the similarity of the models. As an example, we have rewritten a simple version of the lighting system described later and successfully verified through a reachability analysis the absence of a state that violates the assertion: "the internal light is always between 300 and 400 Lux when a user is present in the controlled zone".

The formal verification phase will be studied in greater details in future publications.

IV. Code Generation For Charon

Because we intend to deploy the developed models in embedded devices, we have developed a code-generation tool that transforms Charon programs into embedded Java. This generated code can be used either for the final implementation on a WSAN (Wireless Sensor/Actuator Network), or within an intermediate emulation platform, as will be explained later in the next Section.

The code-generator, called C2EJG (Charon to Embedded Java Generator), has been developed using JavaCC [13]; it auto-translates Charon programs to embedded Java code, which will be used on the Sun-SPOT (Small Programmable Object Technology) sensor nodes within a WSAN [14].

Fig. 2 shows the C2EJG design flow. After modelling the building sub-systems using Charon, a ".cn" file is produced. The code generator then uses this file to generate the corresponding embedded Java code. Finally, we use the NetBeans and Sun-SPOT libraries to produce ".jar" files for each ".java" program.

In the auto-generation process, we distinguish environment- and control-agents. The ".jar" environment files are to be deployed on the sensor nodes. The ".jar" control files will be running on the host station and connected through a base station to ensure wireless communication.

In this section, we outline the main code-generation rules. We start by defining the translation rules for the agent hierarchy, followed by the modes and their behaviour, then the rules to translate types and variables, and finish with the algebraic and differential equations. Through a small example we define the rules to generate embedded Java from Charon with respect to the operational semantics of Charon, as defined in [10].

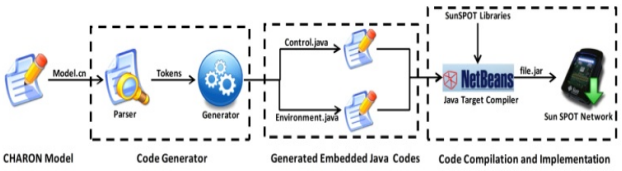

Figure 2: C2EJG Design Flow

A. *Agent Hierarchy*

Code generation follows the hierarchical structure of the model. A primary top-level agent is required in any Charon model, and corresponds in embedded java to the main system. At the second level, we distinguish two types of agents: environment agent(s) and control agent(s). We use an annotation system to help the code generation. Therefore "`//#Environment`" indicates that the agent is used to model the environment whereas "`//#Control`" indicates a modelling of control functionality.

For example, we might have the code fragment

```
//#Control
agent agent_name( ) {...}
```

When the code generator parses "`//#Control`" it considers the next agent as a control agent, and hence generates a java file in order to write the control behaviour in it and follows the inherent agents or modes. If we have the code fragment

```
//#Environment
agent agent_name( ) {...}
```

the following agent is considered as an environment agent and hence it generates the corresponding Java file.

In Charon, agents are considered as concurrent threads; therefore the code generator creates a java file for each top-level agent. These files are concurrently executed through the WSAN. In each file, the inherited agents are translated into concurrent java threads following the generation rules specified in Fig. 3.

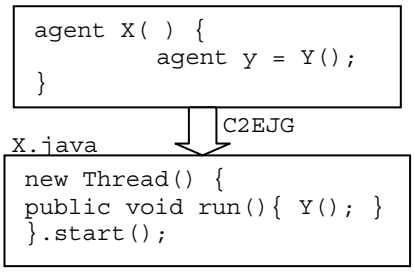

Figure 3: Charon Model Example

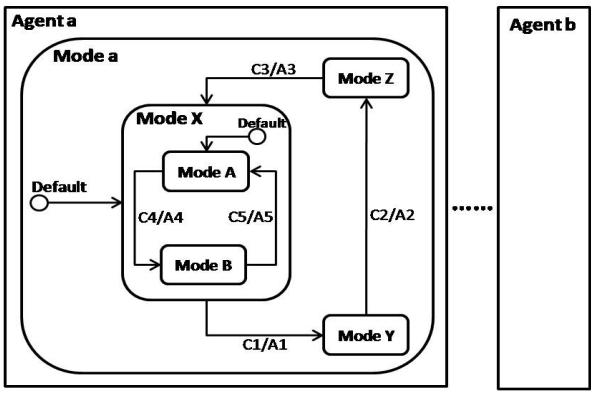

Figure 4: Charon Model Example

B. *Mode Behavioural/Transactional Description*

After translating the agents (as described earlier), we reach the first mode level that can include inherited hierarchy of modes. The first mode (main mode) is translated by a "`while`" loop, using a "`true`" condition in the corresponding Java thread. Regarding the transitions between sub-modes, the code generator translates them into an "`if`" condition on a boolean variable representing the location. The transition guard is translated to a nested "`if`" under the previous "`if`" structure. It conditions the execution of the following actions that are literally re-depicted. The invariant inside a mode is translated into a "`while`" structure with the invariant as a condition within the "`if`" structure. In case no invariant is used the same solution is adopted using the negation of the next transition guard.

Fig. 4 shows a typical Charon model. The translation is illustrated in figures (Fig. 5 and Fig. 6)

```
agent a( ) {
        mode a = a();}
agent b( ) {...}
mode a(){
  mode X = X();
  mode Y = Y();
  mode Z = Z();

  trans from default to X when true do {}
  trans from X to Y when C1 do {A1;}
  trans from Y to Z when C2 do {A2;}
  trans from Z to X when C3 do {A3;}
}
Mode X(){
  mode A = A();
  mode B = B();

  inv {n > 20}
  trans from default to A when true do {}
  trans from A to B when C4 do {A4;}
  trans from B to B when C5 do {A5;}
}
```

Figure 5: Charon Language

```
While(true){
        If(true){x = true;}
        If(X){
                While(n>20){
                  If(true){A = true;}
                  If(A){...}
                  If(B){...}
                }
                If(C1){
                  A1;
                  X=false;   Y=true;
                }
        }
        If(Y){ While(!C2){...}
                If(C2){
                  A2;
                  Y=false;   Z=true;
              }
        }
        If(Z){...}

}
```

Figure 6: Embedded Java Language

C. Variables and Types

The code generator tracks the control agent variables in order to detect the shared variables that are used by both control and environment. These *"write shared"* variables insures the communication between the Sun-SPOTs and the host station. For example:

```
//#Control
agent agent_name() {
        private analog real X;
        private analog real Y;}
//#Environment
agent agent_name(){
        write analog real X;
        read analog real Y;}
```

In this case, there are two shared variables, X and Y, since Y is a *"read"* variable in an environment agent, meaning that it is only used to read its value and is ignored by the code generator.

In general, the code generator provides a designer with the ability to specify the nature of environment to be used in a final implementation, e.g., a software application or simply sensors or actuators. For example, the user can specify, using an annotation system, the type of sensors to be used (Light, Heat, ...). Keywords like "`Light`", when used as a variable name (`write analog real Light;`), indicates the use of a light sensor. The code generator generates the declaration needed to activate the appropriate sensor in the embedded Java file.

D. Algebraic/Differentiation Equations

Regarding the algebraic equations, the code generator copies the same equations in the corresponding "`while`" loop.

```
alge { X == 0; }   →   X = 0;
```

For the differential equations, Charon considers only the first order differential equation. The code generator follows Euler's rule [15], assuming the step in y axis equal to 1, to generate the corresponding Java code inside the "`while`" loop.

```
diff {d(X)== Y}   →   X+ = Y;
```

E. Deployment Issues

The final step of the design is embedding the generated Java code in the hardware platform. The current version of C2EJG supports certain basic network topologies, e.g., a cartelized network topology [16] where all sensors/actuators send/receive the wireless packets only from/to one main controller. To accomplish this embedding phase automatically, we have defined, through an annotation mechanism, notification rules to identify the communication between nodes. For example, we may have

```
//# Enviroment1 -> Control1
//# Enviroment2 -> Control1
//# Enviroment3 -> Control2
//# Control1    -> Control2
```

When a hierarchical network structure is adopted, like in the lighting system example, the code generator considers the shared variables between each connected agents in order to implement the corresponding communication hierarchy. This is possible using the annotation system where the designer can provide the topology requirements.

V. LIGHTING SYSTEM EXAMPLE

In this section, we describe a controller for a Lighting System as a case study. We illustrate the following steps: Charon modelling, code generation of embedded java, and deployment of this code on a WSAN.

The lighting system automatically controls the light intensity in a single or multiple zone(s) depending on user preferences. Given multiple users in a zone, we have to use a preference solver to compute a preference value which integrates the preferences of all users. Fig. 7 shows the use-case UML diagram of the lighting system; the model involves 3 main environments, persons, external light and blinding, as follows.

1. Each person in a zone indicates his light intensity preference.
2. The preference solver receives the light intensity voting, checks the attendance and then calculates the optimal light intensity for the specific zone.
3. The light controller measures the external light coming from outside of the controlled zone and modifies the internal light in order to reach the optimal light intensity.
4. In case when the external light intensity is higher than the optimal light, the controller asks the user to adjust the blinding (Manual/Automatic) margins. If the user selects Automatic control, the controller requests then to identify the mode: Fully Automatic, Semiautomatic. If the user chooses Fully Automatic so the controller controls the blinding in order to reach the best power consumption saving; else the user needs to provide a variation margin of the blinding control and in this case the controller controls the blinding only within the provided margin.

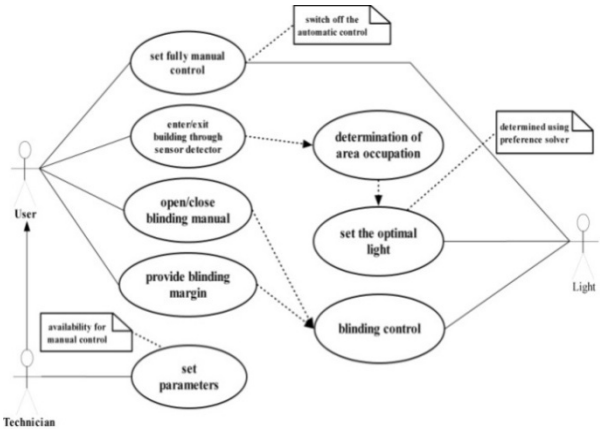

Figure 7: Lighting System Use-case UML

A. Lighting System Modelling

Fig. 8 shows the Charon model for the lighting system, it uses three main environment agents: presence, external light

intensity and blinding variation margin. The output of the three environments allows the controller agent to calculate the internal light intensity.

Fig. 9 shows the changes in the internal light depending on the external light intensity, the optimal light (500 Lux) and the blinding variation margin.

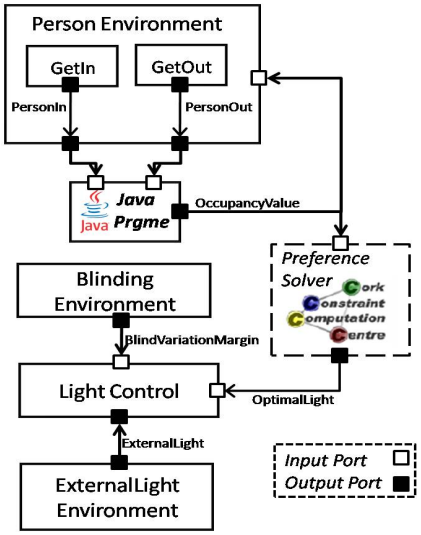

Figure 8: Lighting System Charon Model

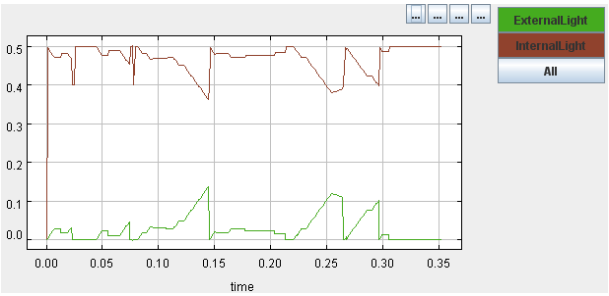

Figure 9: Internal Light vs. External Light

B. Lighting System Emulation

In order to emulate the lighting system, C2EJG has been used to generate the embedded java code equivalent to the Charon model. In accordance to three environment agents and the control agent, C2EJG generated three "env_name.java" files to be deployed on the sensors and one "control_name.java" file to be deployed on the host station, as shown in Fig. 10. In order to test the emulation, the Sun-SPOT Manager [14] has been used to emulate the sensors and communicate with the host station through a base station as shown in Fig. 11.

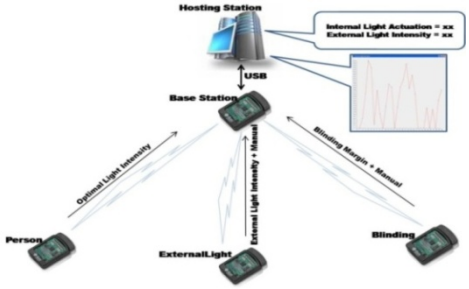

Figure 10: C2EJG Files Deploying

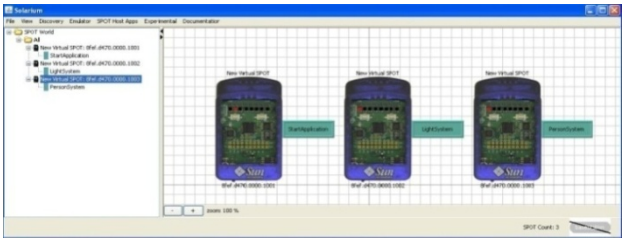

Figure 11: Lighting System Emulation

VI. SUMMARY AND CONCLUSIONS

This article has described a compositional model-driven approach for auto-generating embedded code, which we have used to generate code to improve the energy efficiency of building automation applications. In particular, we described the use of a component-based hybrid-systems modelling framework to generate models to be applied to energy-efficient lighting control for smart buildings. Given a hybrid-systems building model, which can be used for design simulation and verification, our approach can auto-generate embeddable code. We empirically demonstrate this approach using the hybrid-systems tool Charon, and generate embeddable Java code to be deployed on a WSAN. We also provided a detailed description of the code generation steps.

This approach provides some significant, novel capabilities to smart buildings. In particular, we can now deploy distributed control code in a WSAN, given a system-level, centralised system model.

ACKNOWLEDGMENT

This work was funded by SFI grant 06-SRC-I1091.

REFERENCES

[1] http://www.arcat.com/bim/bim_objects.shtml
[2] Labinaz, G. and Bayoumi, M.M. and Rudie, K., "Modeling and Control of Hybrid Systems: A Survey," Proc. of the 13th Triennal World Congress, San Francisco, USA, 1996.
[3] http://rtg.cis.upenn.edu/mobies/Charon/
[4] Suh, N.P., "The Principles of Design," Oxford University Press, USA, 1990.
[5] http://www.mathworks.com/
[6] http://www.dynasim.se/dymola.htm
[7] Gregor Gössler and Joseph Sifakis, "Composition for Component-Based Modeling," in Formal Methods for Components and Objects, Springer Lecture Notes in Computer Science, 443-466, 2003.
[8] Keppens, J. and Shen, Q. "On compositional modeling," in The Knowledge Engineering Review, 16(2), 157--200, 2001.
[9] http://www.iai-tech.org/
[10] Y. Hur, I. Lee, "Distributed Simulation of Multi-Agent Hybrid Systems," 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2002.
[11] Tobias Amnell, Gerd Behrmann et al. Uppaal - Now, Next, and Future. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, Modelling and Verification of Parallel Processes, number 2067 in Lecture Notes in Computer. Springer–Verlag, 2001.
[12] Goran Frehse, "Compositional Verification of Hybrid Systems Using Simulation Relations," PhD thesis, Radboud University Nijmegen, 2005.
[13] T. Copeland, "Generating Parsers with JavaCC," Centennial Books, Alexandria, VA. ISBN 0-9762214-3-8, July 2007.
[14] http://www.sunspotworld.com/SPOTManager/
[15] A Jameson, W Schmidt, E Turkel, "Numerical Solution of the Euler Equations by Finite Volume Methods Using Runge-Kutta Time-Stepping Schemes," AIAA Paper, 1981.
[16] F. L. LEWIS, "Wireless Sensor Networks," Computer Networks, 2002.