A UML-based Design Framework for Time-triggered Applications

Kathy Dang Nguyen P.S. Thiagarajan Weng-Fai Wong
School of Computing
National University of Singapore
{kathyngu, thiagu, wongwf}@comp.nus.edu.sg

Abstract

Time-triggered architectures (TTAs) are strong candidate platforms for safety-critical real-time applications. A typical time-triggered architecture is constituted by one or more clusters. Each cluster consists of nodes communicating with one another via a time-triggered communication protocol. Designing applications to run on such a platform is a challenging task. We address this problem by constructing a UML-based design framework which exposes the essential features of the time-triggered platforms at the UML-level and allows applications to be developed at a more abstract level before full implementation. To support preliminary functional validation, we have constructed a translator by which SystemC code can be automatically generated from UML designs. Our framework enables fast prototyping of time-triggered applications and early design validation. It also supports key design principles of TTAs, such as temporal firewalls and composability.

1. Introduction

Time-triggered architectures (TTAs) [21] are increasingly being advocated as suitable platforms for implementing dependable distributed real-time systems. The key feature of TTAs is a *time-triggered* communication protocol using which the components of a system communicate with each other. Under such a protocol, the communication actions take place at pre-determined time points. The resulting temporal determinacy provides the basis for building robust safety critical real time systems. In particular, TTAs are geared towards automotive, railway and aerospace applications that are safety-critical.

Applications targeted to run on TTAs must comply with the timing requirements imposed by the underlying time-triggered communication protocol. Further, the applications developer must be aware of architectural support that may be available to support the time-triggered communication fabric. Finally, basic design principles such as temporal firewalls (no control signals shall flow across communication interfaces) may need to be enforced in the process of developing applications. Thus one needs a design framework specifically geared towards TTAs. Our main goal here is to present such a framework.

At the top layer we propose a menu of UML-based notations using which applications can be rigorously specified and key parameters of the underlying TTA and communication protocol exposed. UML is now widely accepted in the software engineering community as a common notational standard. It supports object-oriented designs which in turn encourage component reuse. It can be used to provide multiple views of the system under design. It allows standard ways of extending the language to meet the demands of specific application domains. Though it was originally created to serve the software engineering community, UML has been steadily evolving to provide a sound notational basis for developing real time embedded applications [25].

The second major component of our work is a translator that transforms designs developed at the UML level into executable SystemC code for initial functional validation. SystemC allows both applications and platforms to be expressed at sufficiently high levels of abstraction while at the same time enabling the linkage to hardware implementation and verification [17]. Furthermore, SystemC – viewed as a programming language – is a collection of class libraries built on top of C++ and hence is naturally compatible with the object-oriented paradigm that UML is based on. SystemC has the potential to provide a full-fledged description of an execution platform which can serve as the target of a co-design methodology. For these reasons, SystemC is a popular intermediate representation language.

There are a variety of TTAs and associated protocols [30]. Here we have based our ideas on the FlexRay standard. Due to strong backing from the automotive industry [15], it is likely to emerge as the industry standard for a time-triggered in-vehicle communication system. Until recently, work on FlexRay has focused on defining and validating the standard and developing appropriate communication hardware. With this the stage close to completion, one can begin to develop actual applications. In this context, a UML-based backbone such as the one we advocate here can provide necessary level of abstraction and standardization to support reuse.

We expose the key aspects of the FlexRay at the UML level by introducing appropriate stereotypes, tagged values and more importantly, using an activity diagram to specify the static schedule defined by the protocol. Through a disciplined use of ports and interfaces we also enforce the temporal fire wall principle required by time-triggered protocols [11]. This principle demands that only data values but no control signals should flow across communiction/component interfaces of TTA architectures. We also provide preliminary evidence suggesting the composability quality enjoyed by the TTA approach can be reflected at the UML-level. We do so by adding a cruise control application to an initial design that consists of a brake-by-wire application.

Finally, we demonstrate a translation to a standard and popular intermediate representation, namely SystemC [4], for initial validation. A key advantage of our translator is that it automatically synthesizes – using information gathered at the UML layer – a *simulation driver*. This driver mediates between the kernel simulator of SystemC and the generated SystemC code in order to achieve fast simulations.

It is worth pointing out that we do no address the issue of how/whether the generated SystemC code can be used in the software development process. The worst case scenario is that the SystemC code is used only for prototyping and initial validation. Even in this case, it can serve as a valuable design specification for starting the code generation process.

Though we have carried out our work in the specific setting of FlexRay, our approach can be adapted to handle other time-triggered protocols. Based on our previous work [27], we are confident that the framework presented here could be integrated without too much difficulty with event-triggered architectures and applications.

Our Contributions

- Exposing the relevant features the underlying architecture and time-triggered protocol at the UML-level through a suitable choice UML diagram types and fixing their roles. In particular, the structure diagrams are used to display the underlying TTA architecture, restricted behavioral state machines to capture the control flow of application tasks guided by the communication cycles of FlexRay, an annotated activity diagram to display the main features of the static communication schedule.
- Enforcing the temporal firewall principle at the design layer through the restrictions imposed on the usage of ports with proper interfaces.
- The novel use of a simulation driver at the SystemC level to significantly improve simulation speeds and its automatic synthesis by our translator.
- Localizing the changes to be made at the UML-level in order to incorporate a new application and thus achieving composability and reuse.

Plan of the paper In the next section we will describe the design backbone in detail after discussing briefly the FlexRay protocol. Section 3 deals with the automatic translation of the UML models into SystemC. Section 4 presents experimental results based on a case study and in the concluding section we point to possible future research.

1.1. Related work

UML has been proposed to be used in the design process of automotive software applications in different settings. For instance, it has been used for system specification [29, 23] but without mechanisms for validation and synthesis. On the other hand, AnyLogic [13] supports specifications via UML-RT [32] based modeling, generates Java code and executes animated models. Our approach differs from this in that we use standard UML notations with a few extensions. Secondly, we support simulations by automatically generating SystemC code which provides a standard executable intermediate representation. Next we note that AML (Automotive Modeling Language) [6, 5] is also a related UML-based modeling framework that supports multiple abstraction levels. The key difference is that we have a path to an executable intermediate representation that allows functional validation. In addition, our code generation process takes advantages of the special features of TTAs so as to achieve good simulation speeds.

A number of toolsets are currently available that support time-triggered protocols. Typical examples are: TTTech's [2] toolset for the TTP protocol, TTAutomotive's [34], Decomsys' [9] and dSPACE's [33] toolsets for FlexRay. In general, these toolsets deal with the design of TTAs at a more concrete and platform-specific level. An important future challenge will be to bridge the gap between this lower level and the UML-level abstraction we propose.

Model-driven development environments based on SCADE and built on top of TTPPlan and TTPBuild have been proposed [10, 8]. The idea is to model time-triggered software tasks using the GUI of the SCADE tool. The SCADE Code Generator will then generate code that can be run as time-triggered OS tasks. We feel that UML 2.0 notations are an attractive alternative since they are more

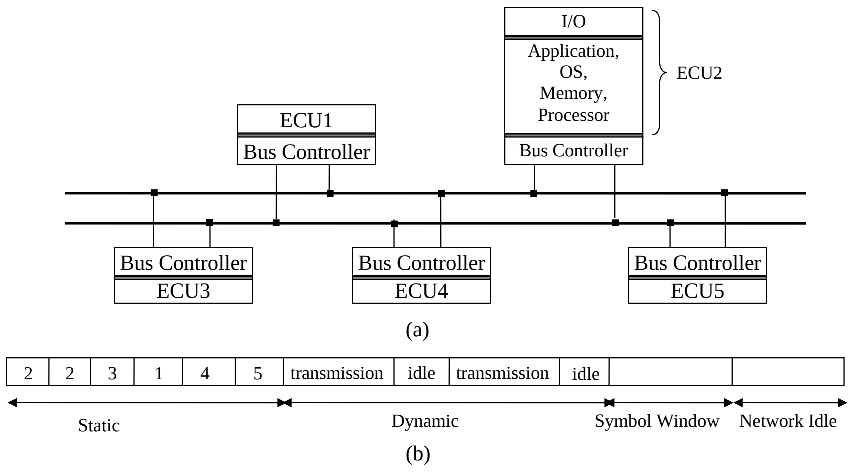

Figure 1. FlexRay Basics

generic, offer a variety of diagram types to deal with both the structural and behavioral aspects of applications. Further, UML, unlike SCADE, is not strongly tied to the synchronous programming paradigm.

STEP-X [26] uses UML notations and targets general automotive applications. It does not cater to the specific needs of TTA-based applications. A UML profile for TTAs is defined in [24] with a proposed mapping from designs developed using this profile to the tools TTPPlan and TTPBuild. We feel however that it is premature to define and impose a full-blown UML-profile on designers at present. Our work shows that the standard notations of UML 2.0 – with a mild dose of stereotypes and tagged values – suffice.

More generally, Giotto [20, 16] offers a software layer for specifying and composing time-triggered tasks. This level of abstraction uses logical time rather than physical time. In contrast, at the UML-level, we use physical time as dictated by the static schedule of the FlexRay protocol. Consequently, in our framework, it will be much easier to relate timing behaviors at the implementation and specification layers. Finally, the AUTOSAR [1] consortium has been working actively on proposing an overall software architecture standard for automotive applications. Our modeling framework is more specific in that it is targeted towards time-triggered platforms.

2. UML-level Modeling

In developing distributed applications that have hard real-time constraints, the time-triggered paradigm advocates a number of design principles [11] to be followed:

- **Temporal firewall:** This design principle requires sender tasks to push data onto the time-triggered communication platform and receiver tasks to pull data from this platform. A sender does not send any control signal directly to a receiver.

- **Global time:** All the local clocks in a TTA cluster must be synchronized sufficiently often to establish the global time of the cluster. The granularity of the global time *g* must be greater than the precision achieved via the clock synchronization mechanism.

- **Composability:** This principle covers several aspects. First, it requires that nodes can be designed independently of each other assuming that the architecture and service have been specified precisely. Secondly, once a node has been validated, integration of the node into a system should not refute the correctness of its service in both the time and value domains unless additional computational resources requirements arise due to the integration. This is called the *stability of prior services*. Finally, the *constructive integration principle* requires that the integration of the $(n+1)^{\text{th}}$ node must not disturb the correct operation of the *n* nodes already integrated in the system.

We now describe the main features of our modeling framework. We focus on our choice of diagrams, their intended role and required usage pattern so that rigorous specifications that adhere to the above principles can be developed. We shall also highlight how the key features of the architecture and the communication protocol are blended into the specifications. This is done in order to facilitate the automatic generation of SystemC code that can be efficiently simulated. As mentioned earlier, we will focus here on a specific embodiment of the TTA principles, namely, the FlexRay standard [14].

2.1. The FlexRay communication platform

FlexRay [15] is a communication protocol for real-time distributed systems. It is implemented on time-triggered architectures in which several ECUs are connected to one or two communication channels as shown in Fig. 1(a). Several other network topologies are possible [14] but we will not consider them here. An ECU consists of a processor, a memory management unit and a communication interface to which a bus controller is attached. Often, an ECU will also have sensors and actuators associated with it. The bus controller mediates between the ECU and the communication channels and implements the key communication-related functionalities of the ECU. These functionalities include the scheduling of message transmission, assembling/deassembling messages, coding/decoding, performing physical access to the busses and clock synchronization.

The ECUs transmit data to each other mainly in a time-multiplexed fashion as dictated by the FlexRay protocol. The protocol executes in recurrent periodic cycles called *communication cycles*. The protocol is implemented using a four level timing hierarchy. Each cycle at the top level consists of a static segment, an optional dynamic segment, an optional symbol window and a network idle time (see Fig. 1(b)).

- Data generated by tasks running on the ECUs are sent and received in the *static segment* in a time-triggered fashion. More precisely, write access to the bus by the ECUs is scheduled according to a fixed *time division multiple access* (TDMA) scheme during the static phase. Consequently, an ECU is granted exclusive write access to the bus at exactly specified time intervals called the *static communication slots*. All slots have the same time duration and exactly one frame is to be transmitted per slot. Further, the order of allocation of slots to ECUs in the static phase is identical for all cycles.

- The *dynamic segment* is in some sense an event-triggered phase where, using a priority scheme, the ECUs can be scheduled to transmit messages of varying time duration measured in terms of *mini slots* in a time-deterministic fashion (see [14]).

- The *symbol window* is used to transmit special messages such as "cluster-wake-up".

- The *network idle time* phase is used to calculate and apply clock correction terms in order to achieve clock synchronization.

At the lower levels of the FlexRay timing hierarchy, static and dynamic slots consist of a fixed number of *macro ticks*. Each macro tick consists of a fixed number of *micro ticks* generated by a local clock. For the purpose of application development, macro and micro ticks are not exposed at the UML level. We also do not consider the symbol window and network idle time segments since they do not directly influence the functionalities of the application tasks. In effect, the applications are modeled with the assumption that clock synchronization is assured by the underlying communication platform.

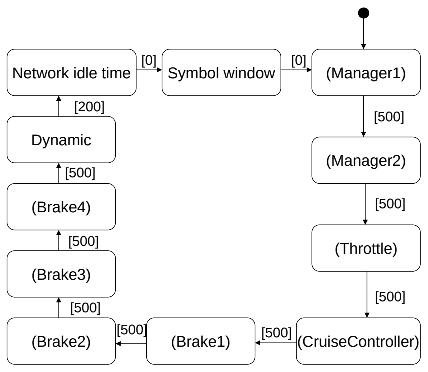

Figure 2. The activity diagram describing the communication cycle

The features of the protocol that we expose at the UML level include:

- The lengths of each segment.

- The number of slots in the static and dynamic segments and their lengths.

- The owner of each static slot (which is also the order in which the ECUs' messages are scheduled in the static segment).

- The ECUs that may send or receive in the dynamic segment.

The above information is captured in a UML activity diagram associated with a usecase depicting how the communication platform is used by the software applications. Such an activity diagram is shown in Fig. 2. In this diagram, an activity node represents a node that is occupying the bus or the phase that the bus is in. A unique token flowing through the diagram determines the current phase of the bus. The condition associated with an activity edge specifies the quantum of time the token must stay in an activity node. In Fig. 2, a token starts at the initial activity. Thus, `Manager1` occupies the bus for 500 micro-seconds before `Manager2` is granted access to the bus. Similarly, `Manager2` is given a 500 micro-seconds slot on the bus, etc. In this example, the dynamic, symbol window and network idle time segments of the FlexRay protocol have been

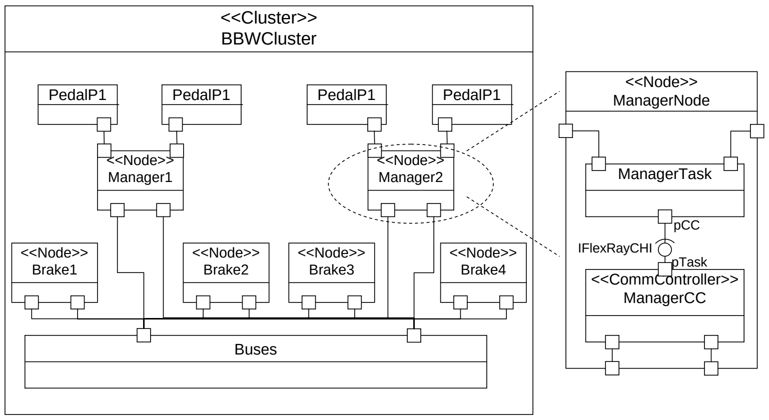

Figure 3. Composite structure diagram of a BBW cluster

assigned 200, 0, and 0 micro-seconds, respectively. The activity node `Dynamic` has a tagged value for the length of each mini-slot inside the dynamic segment and another tagged value which is a boolean expression on `id`s of nodes, describing all the nodes that may send or receive during a dynamic segment. All the above information is required at the application layer to ensure that the tasks are designed to communicate on time. (Due to separation of concerns, we do not consider here how this is achieved). This information is also used by our translator to automatically configure and initialize the communication controller of each node at the SystemC level. We will say more about the communication controllers later in this section.

2.2. Modeling applications

In order to model the software task(s) at each ECU, the structure of the entire time-triggered cluster must be made available. This is because in order to describe tasks and their relationships to the cluster's communication schedule, the application developers need to know on which node a task is located and which nodes the task communicates with are located.

We use *class diagrams* and (*composite*) *structure diagrams* to capture the architecture of the computing and communication infrastructure of the system. A class is an abstraction for those components of a system that have the same behavior and class diagrams are used in the standard way to describe the relationship among the classes. In complex systems, an object of a class may contain other objects of other classes and the relationships between objects may be intricate. UML 2.0 provides structure diagrams to model the internal structure of classes more accurately.

Fig. 3 is a structure diagram depicting the layout and interconnections of a cluster consisting of six ECUs for a Brake-By-Wire (BBW) application. Each object in a structured class is instantiated from other classes. These objects may be connected through association links. The links connecting them show the possible ways they may interact with each other.

An important feature of structured diagrams are the ports with *provided* and *required* interfaces. This allows for a natural description for the provision and use of services by the components of a system. In Fig. 3 the socket notation refers to the required interface `IFlexRayCHI` of port `pCC` while the circle notation refers to the provided interface `IFlexRayCHI` of port `pTask` in the link between object `ManagerTask` and object `ManagerCC`. The `IFlexRayCHI` interface consists of functions for the task `ManagerTask` to call when it wants to send/receive data from the communication controller `ManagerCC`. The required and provided interfaces of two ports connecting to each other must match. In addition, only data-flow can take place across these interfaces and the interfaces include functions to push data to the communication controller to send and pull data from the communication controller to receive. These basic restrictions are imposed in order to satisfy the temporal firewall design principle of TTAs.

Structure diagrams allow hierarchical structures to be captured. For example, one of the classes (instantiated twice in the left half of Fig. 3), namely the `ManagerNode`, has an internal structure shown in the right half of Fig. 3, which is actually the structure diagram of the `ManagerNode`.

Behavior modeling In UML 2.0, an object-oriented variant of statecharts [18] called *behavioral state machines* are used to model the behavior of components in a system. In the present context, application tasks are triggered at specific time instances. Hence, the transitions of our behavioral state machines will be time-outs that denote the amount of time an object has stayed in the source state during which it would have executed the code associated with that state.

The time here is the global time which is assumed to be common knowledge to all the nodes. Guided by these considerations, the behavior of a task assigned to a node will be modeled as a behavioral state machine which at the top level will consist of a *single loop* of states and transitions. The sum of all time-out expressions in this loop must be a multiple of the FlexRay cycle. The time-out expressions associated with the transitions can be constants or arithmetic expressions whose variables must be initiated in the object's constructor. Fig. 4(a) shows an example of the top level of a behavioral state machine. The notations at the top right hand side of the states indicate that the states have actions to be performed on entry. This action can be any C++ (or SystemC) statements which include function calls through ports to transmit or receive data from the communication platform. This state machine models the fact that when a brake actuator object enters the state `starting`, it will execute the state's action on entry. It will stay in this state for 4,000 micro-seconds after which it will take a transition and enter the `dynamic1` state which corresponds to the FlexRay's dynamic segment.

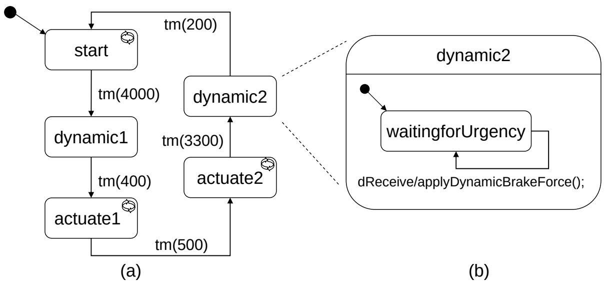

Figure 4. Behavioral state machine of a brake actuator

However, time-triggered transitions in behavioral state machines alone are not enough due to the event-triggered nature of the behaviors allowed in the dynamic segment. To cope with this, we allow the top-level states corresponding to the dynamic segment to have internal states that will capture the event-triggered reactions restricted to the dynamic segment. Fig. 4b shows the internal sub-states of state `dynamic2`. When the system is in the dynamic segment, the brake actuator objects wait for events coming from the communication platform that signal that the car needs to brake immediately. This event can be sent and received only in the dynamic segment and that too only if emergency braking is necessary. If such an event is received, the function `applyDynamicBrakeForce()` is called. Strictly speaking, this is a violation of the temporal firewall design principle as control signals (events) are passed across communication/component interfaces. However, FlexRay violates this principle only within the dynamic segment. Further, even within this segment, the event-triggered signals are generated according to a fixed static priority scheme and users must fix the signals' lengths. Hence temporal determinacy is largely preserved and temporal non-determinacy is confined strictly to the dynamic segments.

In order to eliminate any additional violations of the temporal firewall principle, non-urgent requests or conditions from the environment are not modeled as events in our framework. Instead, they are sent from an object's environment to the object through ports and they are handled in a time-triggered manner, i.e., they take effect only at times specified by the users. Whether these requests or conditions are buffered or overwritten is defined by users in the functions implementing the interfaces of the corresponding ports.

The Design Framework Fig. 5 summarizes our proposed design flow of a time-triggered application using UML 2.0 notations.

For convenience, we add some simple notational extensions – as is allowed by the UML 2.0 standard – through stereotypes and tagged values. The stereotypes of clusters, nodes, and communication controllers are applied to classes as well as to instances of these classes. In addition, tagged values in the activity diagram capture timing details of the physical communication platform. These extensions not only aid the modeling effort, they also contribute to the process of generating SystemC code. They can also be exploited during the implementation stage.

The communication schedule is determined before each task on the nodes are modeled, so the nodes can be developed independently. Our design framework supports composability in the sense that a new application can be smoothly added to an existing model. When a designer wants to extend a cluster with a new application, the following steps must be performed. First we remark that if the new application is to be run on an existing node, then this will basically boil down to a fresh local schedule for that node followed by suitable modifications to the (hierarchical) behavioral state machine describing the execution of tasks on this node. Hence below, we outline the steps needed to add a new node on which the new application will run.

- modify the structure diagram of the cluster class to include a new ECU;
- modify the activity diagram which describes the bus communication schedule;
- model the new node as well as the tasks running on this node for the new application using structure diagrams and a behavioral state machine;
- if necessary, modify the behavioral state machines of the tasks of the existing applications if their schedules

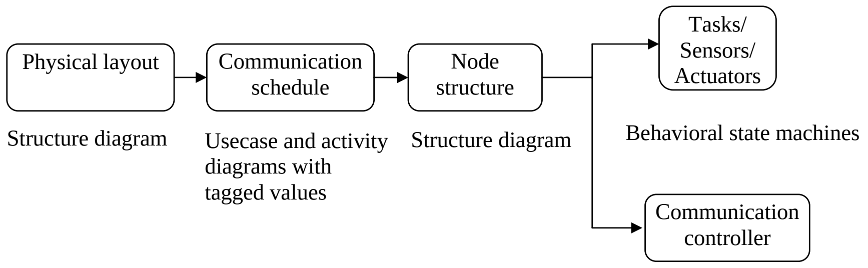

Figure 5. UML-based design flow for TTAs

are affected by the new changes. This modification is restricted only to the time parameter of the timeout transitions. However, timing constraints must be checked to make sure that there is enough time for a task to complete the actions that it is suppose to executed when in a particular state.

The above modifications are well localized and users can easily identify the places in the model which have to be changed.

Finally, a SystemC model of the communication controllers is needed to simulate the system model translated from the UML layer. Users can choose to supply this SystemC model and compose it with the generated SystemC code. Alternatively, they can model the communication controllers at UML level using structure diagrams and behavioral state machines and then use our tool to generate SystemC code for the communication controller automatically. The later approach is faster and easier. In fact we have taken this approach to derive an abstract model of the FlexRay communication controllers at the UML level. These models can be used as a library for application developers who want to develop various applications running under FlexRay. Due to limited space we don't show the UML diagrams of the FlexRay communication controller in this paper. Readers can refer to [3] for these diagrams.

3. Translation to SystemC

We have constructed a translator that automatically converts the UML model of a system into executable SystemC code. We refer the reader to [4] for background information on SystemC. The translation process is not routine and we outline here the main steps as well as the implementation choices we have made in order to obtain fast simulation.

A UML class is translated into a SystemC module. The objects inside a structure diagram of a class will become sub-modules. Initialization codes that create modules and connect them together are generated automatically from the structure diagrams of the structured classes. The initialization codes are then inserted into the respective structured

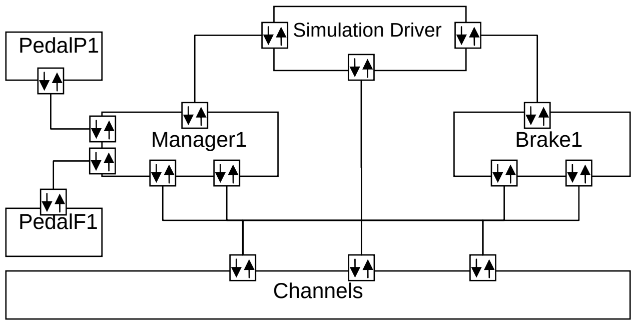

Figure 6. Block diagram for the SystemC generated code

classes' constructors.

Communication through ports defined by interfaces in the UML model is supported naturally in SystemC by the concepts of `sc_port` for ports with required interfaces, and `sc_export` for ports with provided interfaces.

The number of nodes per cluster in a TTA can be large. Our translator automatically configures the SystemC communication controllers by extracting the relevant information from the activity diagram we highlighted in the previous section (see Fig. 2). This process utilizes our UML stereotypes to identify and pass arguments to the nodes, and subsequently to the communication controllers.

The key to fast simulation is how the code for execution of behavioral state machines is executed. The OSCI SystemC simulation kernel is a generic discrete event simulator. It does not take advantage of the crucial property of TTAs, namely, the communication schedule is pre-defined. Thus, if we generate SystemC in the usual way by mapping the behavioral state machine of each class to a SystemC thread, we will get poor simulation speeds. Instead, our translator consolidates a schedule table for a whole cluster. A module called *simulation driver* will execute the model according to this schedule table. In effect, we will have just one SystemC thread residing in the simulation driver to execute all time-triggered actions. To repeat, the simulation driver is generated automatically by our translator. Fig. 6 shows the block diagram of the generated SystemC code from the model in Fig. 3 in which we have eliminated `Manager2` and the three Brakes to improve the clarity of the figure. The simulation driver is connected to each node (and the task on each node) to drive the simulation. The driver is not connected to the two pedal sensors since they are passive modules (there are no state machines associated with them); They just provide functions that the `Manager` can call.

However, additional SystemC threads are unavoidable due to the states corresponding to dynamic segments. Interestingly, we can switch off these threads for the ECUs that don't send or receive during dynamic segment. This information is available via the tagged values of the `Dynamic` activity in the activity diagram.

Our translator can insert codes to print out traces, including state transitions, event notifications and their occurring time.

Implementation We use the Rhapsody tool [28] as front-end for developing our designs. The translation process starts with the Rhapsody internal representation of the UML model from which an XML file followed by an abstract tree of the model is built. After some pre-processing to facilitate the automatic configuration of communication controllers and the generation of the simulation driver, our SystemC templates and the Velocity [35] template engine are used to generate SystemC code corresponding to the UML model.

Although the current front-end of our tool is Rhapsody, our method can be easily adapted to other UML frontends. Any tool that provides a suitable GUI for creating UML diagrams can be used, provided there is a mechanism to generate an XML (or XMI) representation from the graphical interface. This is generally the case for most of the UML tools we are aware of.

XMI is a good intermediate representation for our translator because it is easy to create and extract information from XMI documents. However, XMI itself is not a convenient specification language for system-level designs. In particular it is textual, tedious and error-prone to be used for complex designs. In contrast, UML offers convenient and standard graphical notations for designers to systematically model and document their design.

4. Experimentation

We have experimented with our approach with a number of metrics in mind. The most important one of course is simulation speeds as a function of the number of FlexRay cycles. We also compare our simulation speeds with the simpler thread-based approach in which the simulation driver is not synthesized and instead one SystemC thread per behavioral state machine is created. We also observe the number of lines of code generated by our translator as a (very) rough estimate of the effort saved to create an executable SystemC model.

Table 1. Simulation speed (in ms) of the brake-by-wire application (SD stands for Simulation Driver)

FlexRay cycles	Thread approach	`SD` approach
100	47	8.2
200	99	16.5
500	256	42
1000	497	87
5000	2528	441

A Brake-by-Wire (BBW) application We have created at the UML level an automotive brake-by-wire (BBW) application. The communication components were modeled in accordance with the FlexRay standard. The BBW application was developed based on the material in [19] and [7].

Fig. 3 shows the structure diagram for a cluster. There are six nodes in the system, one for each wheel (brake node) and two manager nodes. Each brake node controls a brake. The manager nodes obtain the force and position applied to the brake pedal from sensors, calculate the brake force that each brake node should apply and send it via the bus to the brake nodes. In turn, the brake nodes send their current status to the managers also via the bus. All these communications are done in the static segment. The dynamic segment is not used here. The FlexRay communication subsystem at each node places data on the bus at the scheduled slots and reads data from the bus when the application needs to.

1,333 lines of SystemC code were generated. At the UML level, we could not gain access to the code corresponding to the algorithms for computing the brake force etc. Instead, we inserted our own simplified code to mimic the functionalities. (This is the also the case for the adaptive cruise controller to be described later.) We simulated the generated SystemC code for varying numbers of FlexRay communication cycles on a PC with a 3 GHz Pentium 4 CPU and 1 Gbytes of RAM. The simulation times, in terms of milliseconds, are shown in Table 1. The simulation times appear to scale well as the number of communication cycles increase. The table also shows the simulation times for the code generated using the threads-based approach. As can be seen and expected, there is a significant gain in simulation speeds when we synthesize a simulation driver which can leverage on the time-triggered nature of the static segments.

We also determined that simulation times obtained via the simulation driver approach are almost the same as the

Table 2. Simulation speed (in ms) of brake-by-wire and adaptive cruise controller applications (SD stands for Simulation Driver)

FlexRay cycles	Thread approach	SD approach
100	77	18
200	146	40
500	389	104
1000	764	196
5000	3214	908

times obtained by hand-creating a SystemC model for the application. Since this is an ad hoc approach, we have not shown this comparison here.

Adaptive cruise control (ACC) To check the extent to which composability is supported, we added an adaptive cruise control application to the BBW cluster.

In this application, once a driver presses the `Set` button, the cruise controller will maintain the car's speed at a setting provided by the driver. It will also maintain a safe distance from the vehicle ahead by adjusting the throttle using the car's current position, speed and its distance from the vehicle ahead. The cruise control will be disengaged whenever the driver steps on the brake pedal or presses the `Off` button. In addition, there is a `Resume` button that allows the car to resume cruise control if it has been disengaged. The `Set`, `Off` and `Resume` buttons (i.e. the corresponding sensors) reside on the same node as the cruise controller.

In our experiment, to implement this new application, two more nodes were added to the BBW cluster; one for the cruise controller and the other for the throttle. This resulted in two more static slots in the static segment of the communication round. One slot is used by the throttle to send the current position of the throttle and the second slot is used by the cruise controller to send a new position that the throttle is instructed to attain. The ACC uses the dynamic segment to handle the emergency situation when there is a vehicle in front of the car at an unsafe distance.

It took the first author around three hours to have the cruise controller application added to the existing BBW cluster, generate the SystemC code for the extended model and debug the new model to get the new application to work properly. The new application could be added quickly because our modeling method, as detailed in the previous section, makes it easy to identify the places where changes have to be made. Further, the SystemC configuration of communication controller is generated automatically. This saves a significant amount of time especially in a system with many nodes. The generated SystemC code is 4,314 lines in total.

Table 2 shows the simulation speeds of the combined application in milliseconds for varying numbers of communication cycles. The dynamic segment is used in this application. Hence the `OR` states corresponding to dynamic segment are mapped onto SystemC threads. So not only are we simulating more nodes compared to the BBW cluster running alone, there are 13 additional SystemC threads due to the use of the dynamic segment. Thus there is a noticeable increase in the simulation times. Admittedly, the new threads are not computationally intensive but as is often the case with SystemC simulations, the speed penalty incurred is due to the context switching that is required in the presence of multiple threads. There are also some slight differences between the simulation speeds of our simulation driver approach and the hand-written code. The simulation driver approach however still gains over the thread-based approach.

5. Conclusions

We have presented a design framework based on UML 2.0 diagrams for applications intended to run on TTAs. We have assumed the underlying time-triggered protocol to be the FlexRay standard. Essential features of the underlying architecture and protocol are captured using the different diagram types and notations of UML 2.0. Our framework complies with TTA design principles such as temporal firewalls and composability. We have built a translator which automatically generates SystemC code for functional verification. Due to the XML-based intermediate representation, the current framework can be easily connected to other tools for additional verification and alternative synthesis.

We have experimented with two different configurations (static segment only, and static followed by dynamic segment)using two standard applications. Our initial experience with this design framework is encouraging. Simulation speed has been optimized by automatically synthesizing a SystemC Simulation Driver which uses a single thread to drive the simulation. We believe this technique will be applicable in other settings as well; in particular, when there is a system-level static schedule involved.

Here, we have focused on systems consisting of a single cluster. However it will not be difficult to extend our framework to handle multiple clusters connected through gateways. It will be important to investigate issues such as (i) developing local schedules in case more than one task is allocated to an ECU, (ii) explicitly modeling bus guardians which prevent violation of the static schedule, (iii) the operating system layer implementing the fault-tolerant clock synchronization mechanisms (iv) the initialization services for waking up a cluster etc.

In our future work, we also wish to explore other time-triggered protocol such as TTP [22] and TT-CAN [12] and include their models in the library of communication platforms for application developers. We also plan to focus on the challenging task of developing a membership service [31] as a core application in order to explore fault tolerance requirements at the UML-level.

In a larger context, one needs backward association mechanisms through which faulty runs (including application's communication's error) obtained at the SystemC level can be traced back to the ill-behaved parts of the UML-level model. The construction of such a mechanism will enable the creation of test benches at the UML level and their verification in SystemC.

References

[1] Automotive open system architecture - AUTOSAR. http://www.autosar.org, 2007.

[2] TTTech Computertechnik AG. http://www.tttech.com, 2007.

[3] UML-based design framework for TTAs. http://www.comp.nus.edu.sg/ kathyngu/UMLSystemC, 2007.

[4] D. Black and J. Donovan. *SystemC: From the Ground Up*. Springer Verlag, 2005.

[5] P. Braun, M. von der Beeck, U. Freund, and M. Rappl. Architecture centric modeling of automotive control software. World Congress of Automotive Engineers, SAE Transactions Paper, 2003.

[6] P. Braun, M. von der Beeck, M. Rappl, and C. Schrder. Automotive software development: A model-based approach. Congress of Automotive Engineers, SAE Transactions Paper, 2002.

[7] M. Bruce. Distributed brake-by-wire based on TTP/C. Master's thesis, Department of Automatic Control, Lund Institute of Technology, June 2002.

[8] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From simulink to scade/lustre to tta: a layered approach for distributed embedded applications. In *ACM-SIGPLAN Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, 2003.

[9] Decomsys website. http://www.decomsys.com/, 2007.

[10] B. Dion, T. Le Sergent, B. Martin, and H. Griebel. Model-based development for time-triggered architectures. In *Digital Avionics Systems Conference (DASC)*, 2004.

[11] W. Elmenreich, G. Bauer, and H. Kopetz. The time-triggered paradigm. In *Proceedings of the Workshop on Time-Triggered and Real-Time Communication*, Manno, Switzerland, Dec. 2003.

[12] T. Fhrer, B. Mller, W. Dieterle, F. Hartwich, R. Hugel, M. Walther, and R. B. GmbH. Time triggered communication on CAN. In *7th International CAN Conference*, 2000.

[13] A. Filippov and A. Borshchev. Daimler-Chrysler modeling contest: Modeling S-class car seat control with AnyLogic. Object-Oriented Modeling of Embedded Real-Time Systems OMER-2 workshop, 2001.

[14] FlexRay Consortium. FlexRay communications system, protocol specification, version 2.1, revision A. 2005.

[15] FlexRay home page. http://www.flexray.com, 2006.

[16] A. Ghosal, T. Henzinger, D. Iercan, C. Kirsch, and A. Sangiovanni-Vincentelli. A hierarchical coordination language for interacting real-time tasks. In *Sixth Annual Conference on Embedded Software (EMSOFT)*, 2006.

[17] T. Grotker. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[18] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, June 1987.

[19] B. Hedenetz and R. Belschner. Brake-by-wire without mechanical backup by using a TTP-communication network. *SAE TRANSACTIONS*, 107(6), 1999.

[20] T. A. Henzinger, C. M. Kirsch, M. A. Sanvido, and W. Pree. From control models to real-time code using Giotto. IEEE Control Systems Magazine 23(1), 2003.

[21] H. Kopetz and G. Bauer. The time-triggered architecture. In *IEEE Special Issue on Modeling and Design of Embedded Software*, volume 91, pages 112–126, January 2003.

[22] H. Kopetz and T. Thurner. TTP - a new approach to solving the interoperability problem of independently developed ECUs. SAE International Congress andExposition, Detroit, MI, USA. SAE Technical Paper 981107, Feb. 1998.

[23] I. Majzik, G. Pintér, and P. T. Kovács. Uml based design of time triggered systems. In *ISORC*, pages 60–63, 2004.

[24] I. Majzik, G. Pintér, and P. T. Kovács. UML based Visual Design of Embedded Systems. In *Proc. The* 7^{th} *IEEE International Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS-2004)*, pages 115–120, Stará Lesná, Slovakia, Apr. 18–21 2004.

[25] G. L. Martin and W. Muller. *UML for SoC Design*. Kluwer/Springer, 2005.

[26] M. Mutz, M. Huhn, and C. Krompke. Model based system development in automotive. *SAE technical paper series 2003-01-1017*, 2003.

[27] K. Nguyen, Z. Sun, P. Thiagarajan, and W. Wong. *UML for SoC Design*, chapter Model-Driven SoC Design: The UML-SystemC Bridge. Kluwer/Springer, 2005.

[28] Rhapsody home page. http://modeling.telelogic.com/, 2007.

[29] M. Rhodin, L. Ljungberg, and U. Eklund. A method for model based automotive software development. In *12th Euromicro Conference on Real-Time Systems*, 2002.

[30] J. Rushby. Bus architectures for safety-critical embedded systems. In T. Henzinger and C. Kirsch, editors, *the First Workshop on Embedded Software, EMSOFT*, volume 2211 of Lecture Notes in Computer Science, pages 306–323, Lake Tahoe, CA, October 2001. Springer-Verlag.

[31] R. Schlatterbeck. Membership service: What it is and why you need one for a safety critical system. *Embedded Intelligence*, 2001.

[32] B. Selic. Using UML for modeling complex real-time systems. In *LCTES '98: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 250–260, London, UK, 1998. Springer-Verlag.

[33] J. Stroop, R. Stolpe, and R. Otterbach. Designing and testing FlexRay systems. *Automotive Electronics II*, 2004.

[34] TTAutomotive Software GmbH. Time-triggered architecture and FlexRay. 2005.

[35] Velocity website. http://velocity.apache.org/, 2007.