MMB 95

Building a hierarchical CAN-Simulator Using an Object-Oriented Environment

Martin Lang, Matthias Stümpfle, Hartmut Kocher[1]

University of Stuttgart, IND
Institute of Communication Switching and Data Technics
Seidenstr. 36
70174 Stuttgart, Germany

As today's systems are becoming more and more complex, simulation is often the only viable way to verify the functionality of a system, or to estimate its performance. In this paper, we will present a flexible general purpose framework for the simulation of complex hierarchical systems. The framework is implemented in C++ and uses high-level abstractions that are closely related to the problem domain. This eases the mapping from a simulation model to an actual simulation program. The framework supports hierarchical decomposition of simulation models into submodels and model components. Model components are strictly encapsulated and communicate with each other using a handshake protocol. This offers the ability to highly reuse standardized model components and quickly create or modify a simulation model using a 'plug-and-play' approach. As an application we used this framework for the simulation of CAN (controller area network) systems. We will show that the hierarchical modelling and the strict encapsulation forced by the framework were real benefits. Different CAN components could be developed seperately and are now available as a CAN part library. Complete systems can now be simulated and evaluated by taking parts from the library and connecting them using the standardized interface from the simulation framework.

1 Introduction

Today's systems are becoming more and more complex. The human mind is unable to comprehend complex systems in their entirety. Therefore, complex systems need to be structured in a way that allows humans to cope with this complexity. This is usually achieved by breaking down the system into a hierarchy of subsystems and modules [2]. In such cases, simulations can help in evaluating different design choices. Simulation programs can be used to verify the functionality of the system as well as to estimate the performance of the target system.

Simulating a complex system is itself a complex task. Therefore, it is important to structure simulation software in a suitable way. The complexity of simulation

---

[1] The author is now with Rational, Pullach im Isartal, Germany

programs can be reduced by decomposing the simulation model into a hierarchy of submodels that can be refined in further steps. Although many parts of a simulation model could potentially be reused across different applications, this is not supported very well in current simulation programs due to strong coupling between model components. Reuse of models and submodels would be an important step towards economic development of simulation programs that could be used to investigate several design choices in a cost efficient way. This is a general problem of software development and is one of the reasons for the so-called software crisis. The object-oriented paradigm promises to improve the situation dramatically. That's why more and more simulation libraries are written in an object-oriented programming language.

Currently, two approaches can be differentiated: on the one hand, simulation environments are mostly targeted at specific application areas. They provide high-level abstractions that are taken from the problem domain. Therefore, they are easy to learn and use. Sometimes, they even offer graphical user interfaces and their own simulation language, e.g., [1, 10]. Unfortunately, most simulation environments cannot be extended by the user, or do not adapt very well to different needs even within the same application domain.

On the other hand, general purpose simulation libraries promise to overcome these difficulties. They are developed using general purpose programming languages. Therefore, they can be extended and adapted easily by the user. Recently, many simulation libraries have been written in object-oriented languages, mostly C++. Unfortunately, most of them focus on implementation issues rather than using abstractions from the problem domain. There is a semantic gap between the low-level abstractions they offer, such as process classes, and the problem-oriented abstractions the user wants. Simulation libraries that are more problem oriented are just emerging, e.g., [9, 12, 13]. However, most systems don't offer support for hierarchical systems. This makes it difficult to implement reusable submodels, and components that can be further refined as the design evolves. As we have already pointed out, these are essential features for simulating complex systems.

This paper describes a general purpose simulation framework that has already been used extensively for the simulation of complex communication systems. It tries to narrow the gap between the problem domain and implementation by using high-level abstractions. Therefore, users can easily map their simulation models to actual simulation programs. The framework can be easily extended because it is written in standard C++. The implementation takes advantage of the latest additions to the C++ language, namely templates and the exception handling mechanism [6]. It uses few basic abstractions and emphasizes a clean software architecture. Hierarchical system decomposition is supported, and it is even possible to write distributed simulation programs [11]. Strict encapsulation with clean interfaces between model components enables massive reuse of simulation models.

In Section 2 we will present the key design issues of the framework. A more detailed description can be found in [7, 8]. Section 3 deals with the modelling and simulation of a CAN system The paper ends with a brief summary.

2 An Object-Oriented Simulation Framework

2.1 Software Architecture

The following section describes the overall software architecture of the simulation framework. Figure 2.1 shows a typical system. Two main parts can be distinguished. The simulation support subsystem contains all components that are necessary to control the execution of a simulation program. It also contains classes that simplify the development of simulation programs, like a modular I/O concept, random number generation, and statistical evaluation. It is further described in Section 2.5.

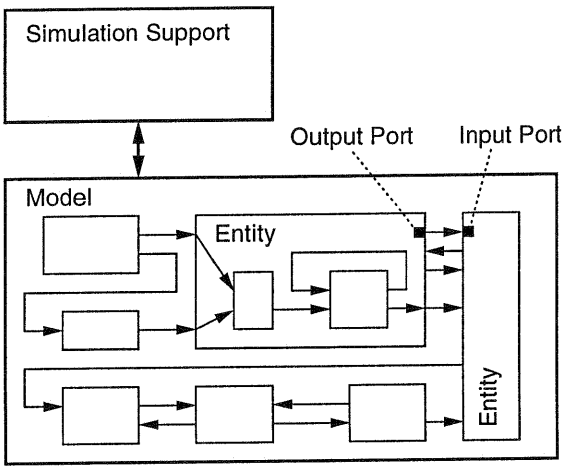

Figure 2.1: System Architecture

The main part of the system is the simulation model. The simulation model can be hierarchically decomposed into submodels and model components. The latter are called entities. Entities communicate with each other by exchanging messages. All messages are derived from an abstract base class that defines some common properties, e.g. the message type. Contents and meaning of a message are user defined. Each entity can evaluate only those aspects of a message which it is interested in.

A port mechanism is used for communication. Transferring messages between entities is as simple as connecting the input and output ports of these entities. A handshake protocol ensures that both entities are ready to exchange messages before they are actually sent. Entities can be seen as black boxes that communicate with the outside world using ports. This strict encapsulation allows separation of the behavior of an entity from the structural arrangement within the model. Therefore, it is easy to insert a new entity between existing entities without modifying the existing ones.

The simulation framework is based on an event-driven paradigm. This approach has been prefered over a process-oriented paradigm because it was clear how a hierarchical event concept should work, whereas there is no suitable definition of hierarchical processes. In event-driven simulations, events are used to plan future activities. Events are entered in a sorted event list and processed later. The meaning of an event depends on the entity that generated it. Events are passed to the entity for processing. The entity might process the event itself, or may pass it on to its parent entity. This supports hierarchical processing of events.

A model entity is a special entity that has a built-in event list. Usually, the model entity stays at the top level of the simulation model hierarchy. Since a model may be composed of more than one submodel, the framework supports more than one event list. The submodels are responsible for synchronizing distributed event lists.

The following sections describe the above-mentioned mechanisms in more detail.

2.2 Model Components

A simulation model can be seen as a network of model components, which we call entities. All entities are derived from a base class *TEntity* that defines the common properties of all entities, such as naming, and methods for dealing with ports and events.

Decomposing a model into a hierarchy of entities is an important means to reduce overall complexity. Therefore, each entity can contain other entities internally. If this principle is applied recursively, it leads to a tree structure of entities with the model entity as the root. Each entity has a reference to its parent entity. Figure 2.2 depicts a simple queuing model that shows two network nodes in a communication system. Each node consists of an input queue and a server entity. Figure 2.3 shows the resulting object tree using the notation of [2, 3]. The mapping from the components of the simulation model to the implemented entities is straightforward.

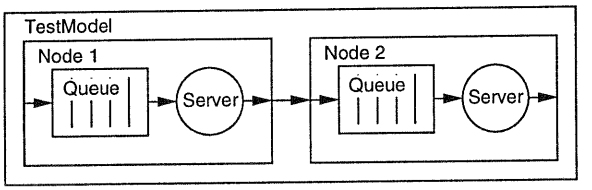

Figure 2.2: A simple simulation model

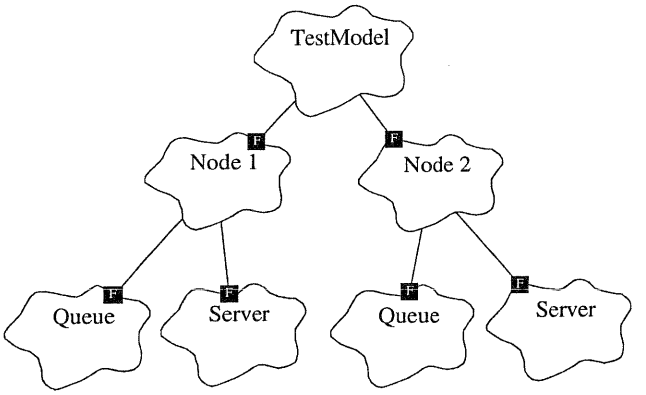

Figure 2.3: Object diagram of the model

An interesting part is the interaction of entities. In order to ease reuse of entities, the coupling should only be as close as absolutely necessary. During initialization, each entity gets a reference to the parent entity. Because it does not know the exact type of the parent, it can only use services that are already defined in the base class *TEntity*. Because of polymorphism, the behavior of those services still depends on the actual type of the parent entity. That way, many services can be delegated to parent entities without knowing the structure of the containment hierarchy. Whereas child entities may only use anonymous services of the parent entity, parent entities do know all child entities. Therefore, they are allowed to call all methods of their children directly without sacrificing encapsulation.

2.3 Port Concept

The port concept is used to pass messages between entities. To improve type safety, input and output ports are distinguished. All connections are unidirectional point-to-point connections between two ports. Ports are registered with their owning entity during construction. Ports may be defined as member objects of the owning entity, or they may be created dynamically. The latter is useful for general purpose components like multiplexers, where the number of input ports depends on the simulation model. Two ports can be connected by calling the *Connect* method of the *TEntity* class. This method also checks if a connection is legal.

All messages must be derived from a common base class *TMessage*. The content of a message depends on the simulated problem and can be defined in derived classes.

Messages are passed between ports using a handshake protocol. This is shown in an object diagram in Figure 2.4. After an entity notifies an output port that a new message is available the port calls the *MessageIndication* method of the corresponding input port. The receiving entity can then decide if it is willing to accept the message. It can do so by calling the *GetMessage* method of the port. If it is unable to receive a message in the current state, it may simply ignore the call. In this case,

the sender is blocked. Later, the receiving entity may call *IsMessageAvailable* to find out if there are messages to receive, and call *GetMessage* to actually receive them.

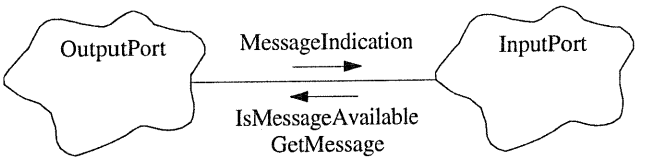

Figure 2.4: Handshake protocol between ports

This simple protocol adds a lot of flexibility because entities do not have to know how messages are created or utilized, or if the receiving entity is in a state where it can accept new messages. Because of this loose coupling between ports, it is always possible to insert new entities between existing ones without influencing the way messages are transferred. This greatly enhances opportunities for reuse because many models can be changed by simply inserting new entities. Other port schemes that do not implement flow control mechanisms are less flexible because messages cannot be blocked between entities, e.g. [9].

Because entities know their ports, they can reference them easily, and call port methods directly. Ports also know their owning entity, but they do not know which method to call in case of a message indication. Especially, if an entity has more than one port, all ports would call the same method. Therefore, a class *TMessageHandler* was introduced to decouple entities and ports. A message handler may either handle a message directly, or delegate it to the owning entity. Template based message handlers allow to call arbitrary methods of the entity class in a type safe manner.

Although it would be possible to create special entities to count messages, or manipulate them, the overhead would be prohibitive. Message filters that can be installed in every port offer a more elegant solution for these kind of problems. If filters are installed in a port all handshake calls between ports are first dispatched to all filters before they are sent to the port or a message handler. Again, template based filters that are derived from the *TMessageFilter* class can be used to delegate these calls to other classes in a type safe manner. By installing two message filters that work together, message transfer times between any two ports can be evaluated.

2.4 Event Handling

Event processing is the core of any event driven simulation program. Normally, events are stored in an event list that is sorted by event time. When the current simulation time matches the event time, the event is processed. Additionally, the concept presented here supports hierarchical event handling.

All events must be derived from a common base class *TEvent*. Similar to the usage of message handlers and filters, users can derive their own event classes. Template based classes can be used to delegate event processing to arbitrary classes, e.g. the entity class that created the event.

The *PostEvent* method of class *TEntity* takes an event and the event time as parameters. Once an event has been handed over to an entity, the entity tries to find a suitable event handler that is willing to handle the event. Events have different types. Event handlers can either handle events of one specific type, or of all types. Event handlers may be installed in any entity. First, entities search their own handlers to find one that is willing to handle the event. If none is found, the *PostEvent* method of the parent entity is called. This technique is applied recursively until either a suitable handler is found, or the root of the model hierarchy is reached, which would cause an exception.

Event handlers may either intercept events, or pass them on to the next higher level of the entity containment hierarchy. An event list is just a special case of an event handler that stores events and processes them later. In order to intercept events not only when they are posted, but also before they are processed, handlers may add an embedded event to the current event. When the *ProcessEvent* method of the original event is executed, all embedded events are processed before the original event.

With this concept, the parent entity is able to manipulate all events of their child entities without modifying the children. Therefore, this scheme can be used for conventional event processing, and for anonymous communication between entities in different hierarchy layers.

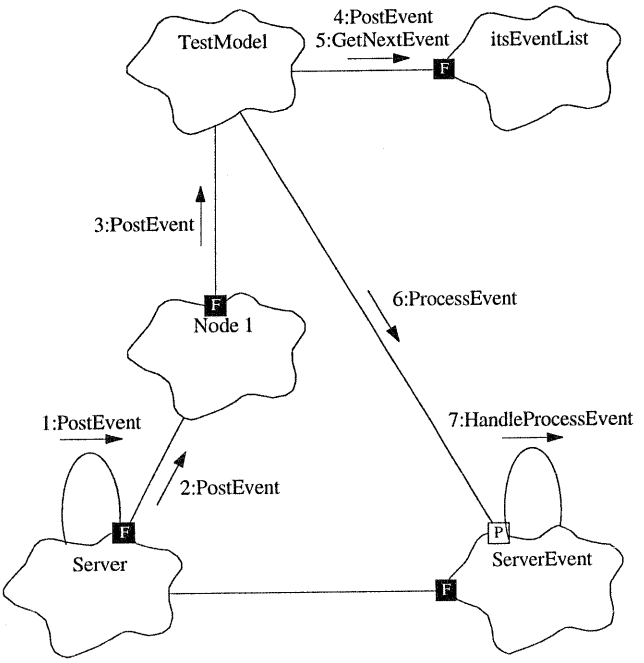

Figure 2.5: Hierarchical event handling

Model entities are special entities that include event lists. The *PostEvent* method of a model simply inserts the event in the event list. The simulation control class and the model entities work together to retrieve the next event in the event list, which is processed by calling its *ProcessEvent* method. The scenario in Figure 2.5 is based on the model that was shown in Figure 2.2. It shows a scenario where an event is posted by the server. Because no event handlers are installed in the server and network node entities, each entity delegates the *PostEvent* method call to its parent entity. That way, the event follows the hierarchy up to the model entity, which inserts the event in the event list. Later, the event is retrieved and executed. Because submodels may be installed at any part of the hierarchy, more than one event list may be used. If event lists are distributed, the models are responsible for synchronizing them.

2.5 Other Concepts

The execution of simulation programs must be controlled. The main tasks are initializing data structures, processing input parameters, running the simulation, collecting and printing the results the user is interested in, and finally stopping the simulation. The presented framework supplies a set of classes that provides a flexible environment for simulation control. It can easily be customized by overriding selected methods.

Additionally, the framework provides a hierarchy of random number generator classes that implement some of commonly used algorithms [5]. Based on the random number generators, a hierarchy of different distributions ranging from simple uniform, binomial, or poisson distributions, up to sophisticated state-dependent models for video sources that are needed for simulating broadband communication networks is available.

To simplify the collection of sample data during a simulation run, a number of meter classes are provided. These meter classes can easily be connected to any port of an entity. Currently, two types of meters can be distinguished, meters that simply count messages, and meters that measure transfer times, i.e. the time a message needs to travel from one point of the model to another. For statistical evaluation of the measured data, a hierarchy of different statistical classes is available.

Finally, the framework supports the developer by providing I/O mechanisms. Currently, input parameters are read from a file by a keyword based parser. If desired, it can easily be expanded by a graphical layer. The output of simulation results can be controlled through usage of styles. Actual simulation results are marked by keywords, and are replaced with current data in the output stream, similar to mail merge applications. Therefore, there is no need to modify code in order to print results in a different format.

3 Simulation of a CAN-System

In the following section, we will describe some insights that we gained while using the library for the simulation of a CAN-System. After a short introduction into the CAN protocol we will focus on the design of the simulation using the introduced library framework.

3.1 The CAN protocol

The CAN protocol [4] was introduced by Robert Bosch GmbH in 1987 to allow cheap intra-car communication over a serial bus system. The protocol regulates the media access by applying CSMA/CA with CA abbreviating „Collision Avoidance".

To allow an arbitration process the channel knows two different logical states: „0" represents the dominant state and „1" stands for the recessive state. If two stations start sending simultaneously, each bit on the bus is compared to what the station was willing to send. If any difference is detected, i.e. the station is trying to send a recessive bit whilst another station is sending a dominant bit, the station that looses the arbitration stops sending. This results in having no latency for the winning station.

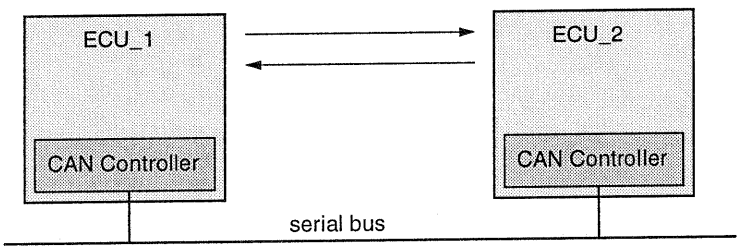

Figure 3.1: Simple two-node CAN System.

CAN systems typically consist of several electronic control units (ECUs) that are connected by the serial bus. Inside an ECU you will find a microcontroller with the application running on it and a CAN controller with the implemented protocol.

3.2 Elements of the CAN Model

The simulation of a system has to guarantee the functional equivalence to reality. The CAN protocol therefore allows the decomposition of a controller net into two basic parts: the controllers with their duty to manage messages with different priorities, and the bus with its centralized arbitration and routing functionality. The coresponding simulation models are depicted in Figure 3.2 and Figure 3.3, respectively.

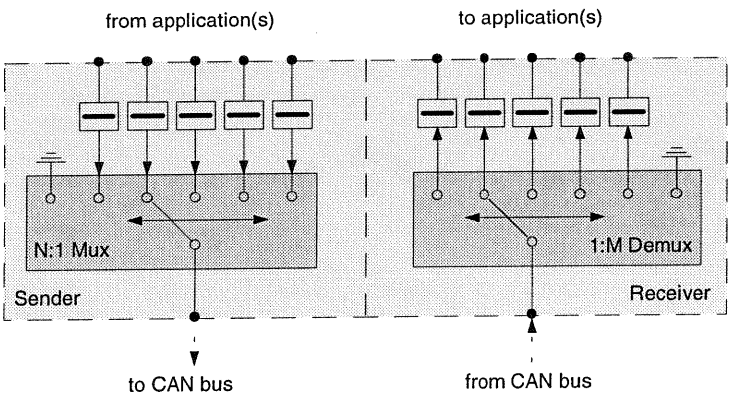

Figure 3.2: CAN Controller Model

Controllers consist of a sender part and a receiver part. They provide buffers for messages that have to be sent over or have been received from the bus, respectively. To each buffer a certain message identifier is attached. The controller decides, by comparing identifiers, into which buffer a message is written.

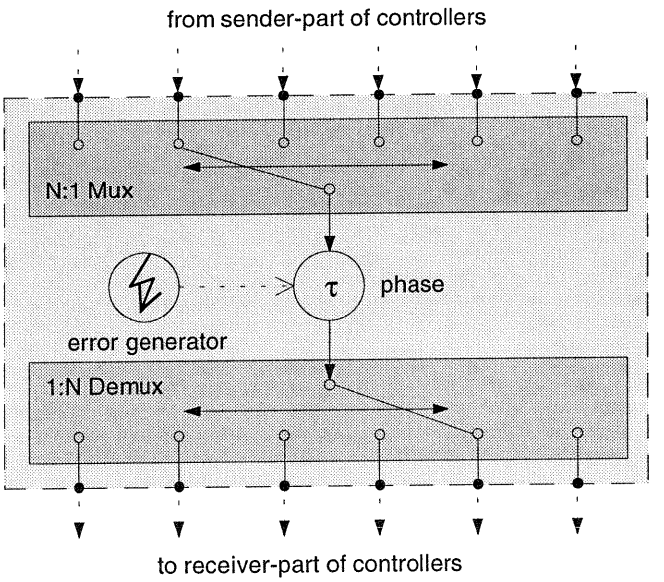

Figure 3.3: CAN Bus Model

The bus model contains the only service phase of the whole model. It represents the access of a message to the bus. Additionaly, an error generator may be switched

to the system. Bus access is gained via a N:1 multiplexer and the routing of messages is done by a 1:N demultiplexer.

3.3 Building a hierarchic simulation model

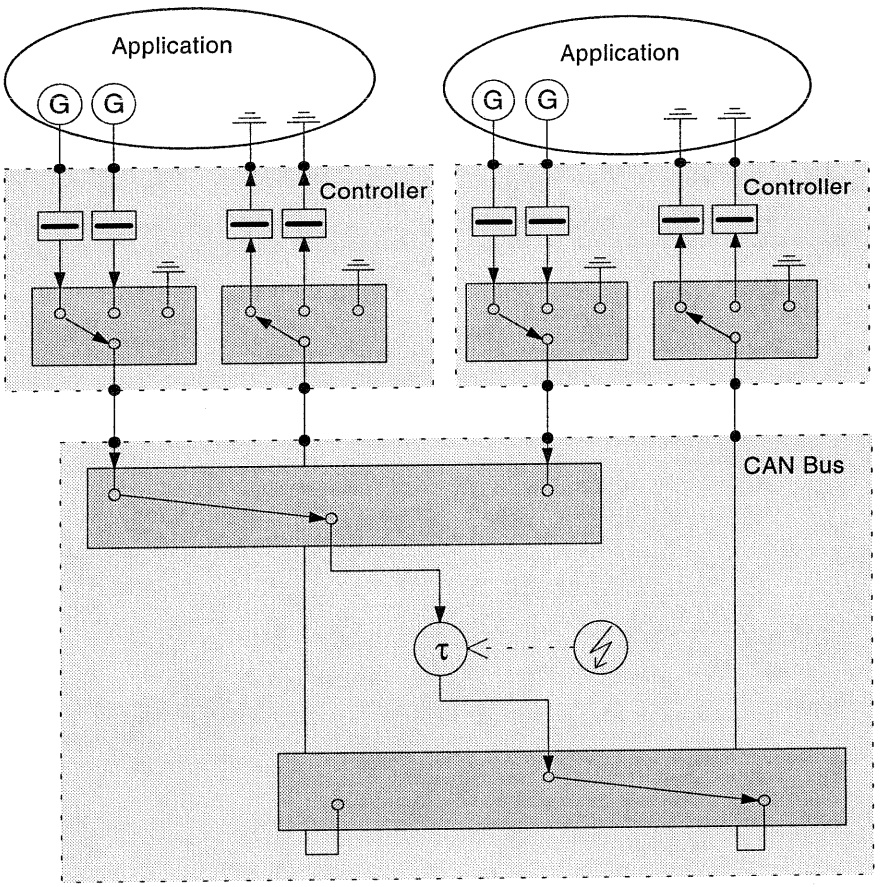

Figure 3.4: A complete CAN-Model with two nodes.

The mapping of the model to the simulation is straightforward. A hierarchy of entities can be derived directly from the model. During the development of the program another great benefit of the object-oriented approach became obvious. The library supports an incremental development process. Due to the encapsulation of the entities and the framework which is provided by the library, it is always possible to build a reduced model (single controllers, the bus) which can be tested separately. Later, individual entities are combined to hierarchical entities, and their ports are

connected. This has the advantage that an executable program is available during every stage of the development process. The need to integrate a large and complex system in one step does not exist. Since the individual entities are already tested, testing of the whole program can be reduced to validating interactions between entities.

During the implementation of the program, we could easily reuse the framework provided by the library. The queues, service phases and generators could either be used directly from the library, or had only to be slightly modified (like the Mux and Demux classes). These modifications could easily be accomplished by deriving new classes from the library entities, and overriding specific methods.

An extension of the presented model by a communications software layer is currently under development. Therefore, we see the CAN model as a basic abstraction and use the input-ports of the controllers as interface to the overlaying software. The developper of the software model does not need to know how the CAN model works, he just „plugs" the CAN model to his model and then is able to run a whole system simulation.

4 Summary

In this paper, we presented a flexible object-oriented simulation framework. One of the main concepts of the framework is complete support for hierarchical decomposition of simulation models including hierarchical event processing. This enables direct mapping of complex simulation models to simulation programs, and also supports iterative refinement of models as the design evolves. The chosen abstractions are very flexible and can even be used for distributed simulation [11].

The presented framework was used for the simulation of CAN systems. Due to the strict encapsulation and the clear software architecture reuse of model components and whole submodels was supported. Iterative development of complex simulation programs has been encouraged because model components could be implemented and tested separately. Modifications were easy to implement because they were just another incremental step in the development cycle.

5 References

[1] Belanger R.F.: *MODSIM II: A Modular, Object-Oriented Language*, Proceedings of the 1990 Winter Simulation Conference, New Orleans, LA, 1990, pp. 118-122.

[2] Booch G.: *Object Oriented Analysis and Design with Applications*, 2nd Edition, Benjamin Cummings, Redwood City, CA, 1994.

[3] Booch G.: *The Booch Method: Notation*, Rational, Santa Clara, CA, 1992.

[4] Robert Bosch GmbH: *CAN Spezifikation 2.0*, Stuttgart, 1991

[5] L'Ecuyer P.: *Random Numbers for Simulation*, Communications of the ACM 33, no. 10, 1990, pp. 85-97.

[6] Ellis M.A., Stroustrup B.: *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Company, Reading, MA, 1990.

[7] Kocher H.: *Design and Implementation of a Simulation Library Using Object-Oriented Methods*, Dissertation, Institute of Communications Switching and Data Technics, University of Stuttgart, Germany, 1993. [In German].

[8] Kocher H., Lang M.: *An Object-Oriented Library for Simulation of Complex Hierarchical Systems*, Proceedings of the Object-Oriented Simulation Conference (OOS'94), Tempe, AZ, 1994, pp. 145-152.

[9] Mak V.W.: *DOSE: A Modular and Reusable Object-Oriented Simulation Environment*, Proceedings of the SCS Multiconference on Object-Oriented Simulation, Anaheim, CA, 1991, pp. 3-11.

[10] Melamed B., Morris R.J.T.: *Visual Simulation: The Performance Analysis Workstation*, IEEE Computer 18, no. 8, 1985, pp. 87-94.

[11] Necker T.: *An Object-Oriented Library for Distributed Simulation*, Proceedings of ASIM'94, Stuttgart, 1994, pp. 235-240. [In German]

[12] Vaughan P.W., Newton D.E., Johns R.P.: *PRISM: An Object-Oriented System Modeling Environment in C++*, Proceedings of the SCS Multiconference on Object-Oriented Simulation, Anaheim, CA, 1991, pp. 32-39.

[13] Zheng Q., Chow P.: *EXsim: A General Purpose Object-Oriented Environment for Descrete-Event Simulations*, Proceedings of the 1993 Western Simulation Multiconference, La Jolla, CA, 1993, pp. 15-21.