From an Abstract Object-Oriented Model to a Ready-to-Use Embedded System Controller

Stanislav Chachkov
Software Engineering Laboratory
Swiss Federal Institute of Technology
1015 Lausanne, SWITZERLAND
Stanislav.Chachkov@epfl.ch

Didier Buchs
Software Engineering Laboratory
Swiss Federal Institute of Technology
1015 Lausanne, SWITZERLAND
Didier.Buchs@epfl.ch

Abstract

We present an example of a construction of an embedded software system - a controller - from the formal specification to executable code. The CO-OPN (Concurrent Object Oriented Petri Net) formal specification language is used for modelling the controller and the associated hardware system with the inherent limitation of its physical components. CO-OPN formal language is based on coordinated algebraic Petri nets. The CO-OPN model can be used to verify some properties of the controller in the concrete physical environment. This is achieved by constrained animation of the valid prototype produced by automatic code generation. The possibility to incrementally refine the generated code can be used to obtain a more efficient implementation.

1 Introduction

Developing embedded reactive software systems needs modelling tools that can capture the properties of the system to develop as well as the structure of the interactions between the software and its environment.

In this paper, we present a formal framework for the development of embedded systems from the modelling phase to the implementation. The approach we propose has adopted the object-oriented paradigm as a structuring principle. We have devised a general formalism which can express both abstract and concrete aspects of systems. This approach is called Concurrent Object-Oriented Petri Nets (CO-OPN)[3][1].

An implementation can be automatically produced from the abstract CO-OPN model. We shortly describe our code generation techniques. Apart from the problem of producing programs that respect the abstract models and their particularities (non-determinism, atomicity of the events, modularity induced by Object-Oriented structure,...), we are particularly interested to match the usual programming principles of the target language. For our purpose they will be the Java notion of component architecture, the Java Beans model[10]. In order to cope with incremental refinement of the automatically generated prototype [6], it is important to be able to interconnect the produced components with other components or with standard libraries (for example user interface libraries). This will be achieved by fulfilling the rules of the component model of target language and by transparently introducing the support for transactions and non-determinism.

This paper presents three new aspects of CO-OPN research. First, a modelling approach that includes both logical (software) and physical elements of the system in one model. Second, the code generation techniques that maps CO-OPN components to target language components. Third, the implementation of an automatic code generator.

Our work lies under the context of executable specifications. Since CO-OPN formalism has formal semantics based on logical programming and Petri Nets, various existing techniques of verification and validation can be applied. This is the major advantage of our approach.

We will first briefly explain how to start from a diagram establishing the interconnection of the application to the outside world elements and how to produce, step by step, a model that will be used to derive automatically a program. An example of a drink dispenser will illustrate our approach. The complete specification and supporting tools can be found at: http://lglwww.epfl.ch/Conform/Coopn-Tools

The paper is organized as follows. In Section 2, we present the drink dispenser and discuss the models that we can produce using CO-OPN. In section 3, we present our basic mapping method for generating OO code from CO-OPN and discuss implementation problems. In section 4, we discuss how to use the produced software for validation and verification purposes.

2 Modelling with CO-OPN

CO-OPN is an object-oriented modelling language, based on Algebraic Data Types (ADT), Petri nets, and IWIM coordination models [5]. Hence, CO-OPN concrete specifications are collections of ADT, class and context (i.e.

1074-6005/01 $10.00 © 2001 IEEE

coordination) *modules* [9]. Syntactically, each module has the same overall structure; it includes an *interface section* defining all elements accessible from the outside, and a *body section* including the local aspects private to the module. Moreover, class and context modules have convenient graphical representations which are used in this paper, showing their underlying Petri net model. Low-level mechanisms and other features dealing specifically with object-orientation, such as sub-classing and sub-typing, are out of the scope of this paper, and can be found in [1] [3].

2.1 The Drinks Dispenser

In order to present a concrete example of modelling, we chose to study a simple but non-trivial kind of reactive system: a Drink Dispenser Machine (DDM) controller. The aim of this study is to elaborate a CO-OPN specification of the dispenser and to explain how to generate the controller program. Moreover, we would like to validate and verify properties of the DDM Controller by using a model including hardware parts, and also to verify the DDM in the context of human interaction. Therefore, our model has three levels: Controller, Machine and Human Interaction (not detailed here).

First of all, we will explain the steps that can lead to the CO-OPN model of the DDM controller. In figure 1, the reader can have a feeling how this machine usually works. Firstly, the customer has to introduce coins in the machine (1), then s/he has to select the desired drink (2) and finally the dispenser supplies the drink via the bottom drawer (3).

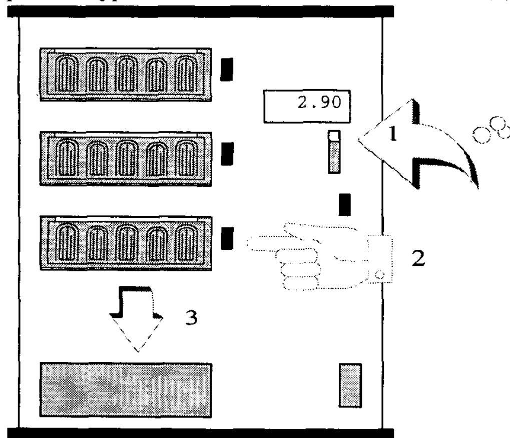

Figure 1: The drinks dispenser machine (real life view)

For such a system we are interested in a first approach to determine the components and the various interactions between them. The main concepts used to express the structure and the behavior of the system are:

- a coordination model for describing the relations between the system components,
- object orientation for the structure and content of the system,
- causality relations for the dynamic aspects that must be reflected with non-deterministic and concurrent behaviors.

The controller is a program that coordinates the activities of DDM components: money box (MB), containers of drinks (DC), LCD display, drink selection buttons (SB) and "return coins" button (RB). The controller receives messages from and sends commands to the equipment (figure 2).

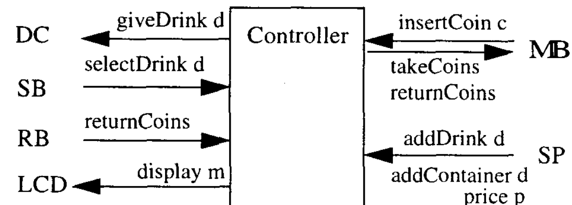

Figure 2: Protocol of Controller

When adding drinks the service person (SP) communicates with the controller directly by notifying `addDrink` and `addContainer` events.

The Money Box is composed with two different collectors of coins: the first one keeps coins until the customer chooses a drink or presses the "return" button, and the second one receives coins for distributed drinks.

Drinks are stored in containers. Each container is associated with a kind of drink and a price. Initially, the DDM is empty. The service person installs containers and adds drinks to them.

The DDM equipment introduce some constraints: the money box has limited capacity, the number of slots for DC and the quantity of bottles in DC are also limited.

2.2 CO-OPN Coordination model

In this section the various concept of CO-OPN will be introduced in the necessary order for the modelling of the DDM. As we use a kind of top-down strategy for modelling, we will first start by presenting the interface of the machine given by the top-level coordination entity called DrinksDispenser context.

A useful approach for building systems composed of many computing entities is to use the high-level concept of *coordination programming* [12]. The term *coordination theory* refers to theories about how coordination can occur in various kinds of systems. We state that coordination is *managing dependencies* among activities.

Taking a step further in this direction, it appeared that

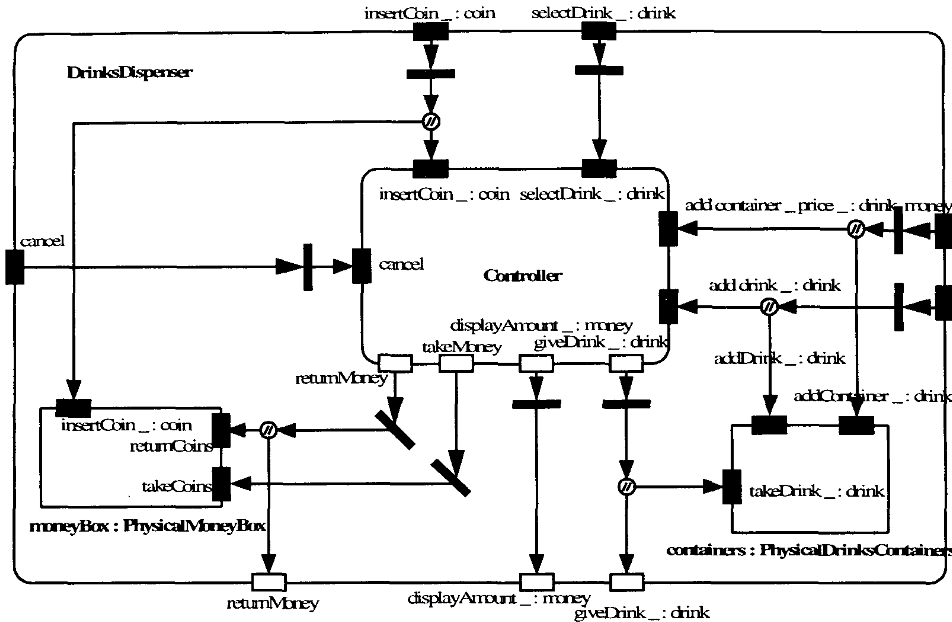

Figure 3: The drinks dispenser machine model

coordination patterns are likely to be applied since the beginning of the *design phase* of the software development. This fact gave birth to the notion of *coordination development* [5]. This process involves the use of specific coordination models and languages, adapted to the specific needs encountered during the design phase as expressed in the drinks dispenser.

Due to their intrinsic nature, IWIM (*Idealized Workers, Idealized Managers*) coordination models [13] are particularly well suited for the coordination of software elements during the design phase [4]. The coordination layer of CO-OPN [5] [3] [4] is a coordination language based on a IWIM model, suited for the formal coordination of object-oriented systems. CO-OPN context modules define the *coordination* entities [11], while CO-OPN classes (and objects) define the basic *coordinated* entities of a system. CO-OPN allows one to cover the formal development of concurrent software from the first formal specification up to the final distributed software architecture [1].

2.3 Coordination with Contexts

In figure 3 we can see the `DrinksDispenser` machine including the `Controller` context and a model of the physical components `PhysicalMoneyBox` and `PhysicalDrinksContainers`, with the input events (or the provided services called methods - black rectangles) and output events (or the required services called gates - white rectangles).

The `Controller` context contains sub-components that interact to provide the controller behavior. The controller sub-components are two objects which are instances of the classes `MoneyBox` and `CentralUnit` as depicted in figure 4. In this picture the oriented arcs between methods or gates are used to define strong synchronization between events. In CO-OPN, it means that the firing of synchronized events is strongly synchronous and atomic. The synchronization of entities in a context is an oriented couple of synchronization expressions of the form: `synchro` **`With`** `synchro`.

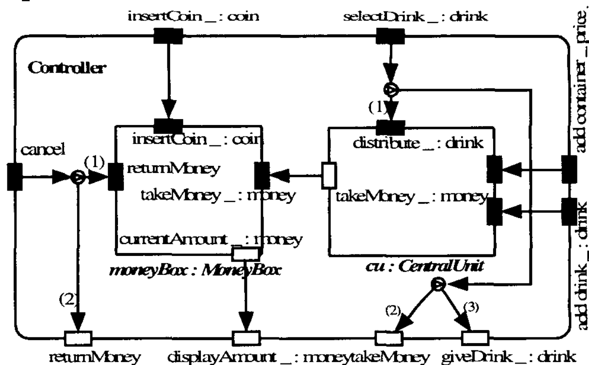

Figure 4: The static components instances inside the Controller context

Synchronization expressions are built with the simultaneity, the sequence and the non-determinist operators. The `MoneyBox` component is devoted to managing the money of the dispenser. The `CentralUnit` component transmits the requests to the specific drink container. The container objects, each containing one kind of drink, are instantiated

by means of the addcontainer _ price _ method that must be called before the add drink _ method. In figure 3, we can observe that the physical components are duplicated by logical components in the controller using simultaneous synchronization (circle with // inside). For instance, the MoneyBox logical component is duplicated by PhysicalMoneyBox and CentralUnit by PhysicalDrinksContainers.Pushing a drink selection button sends the event selectDrink d to the controller. If DDM contains a drink d and enough money then commands takeMoney and giveDrink d are emitted, otherwise nothing is done.

Let us explain how this behavior is defined by the Controller context. In order to express this behavior we link selectDrink d event to a synchronization: cu.distributeDrink d .. takeMoney .. giveDrink d. The event selectDrink will complete with success if and only if each of three sub-events are completed with success. Controller internal event cu.distribute d will check if there is a drink d in DDM and enough money in moneyBox. If those preconditions are satisfied the system will evolve by removing one drink d and moving the money from intermediary collector to the permanent one. In the case of success takeMoney..giveDrink d events will be emitted by controller to external connected equipment (MB and DC).

Before explaining the components, we will quickly give an outline of the way values can be defined in CO-OPN, using algebraic data types.

2.4 ADT Modules

CO-OPN ADT modules define data types by means of algebraic specifications. Each module describes one or more sorts (i.e. names of data types), along with generators and operations on these sorts. The properties of the operations are given in the body of the module, by means of positive conditional equational axioms. For instance, figure 5 describes the ADTs defining two sorts, coin and drink, defined by several generators and one operation, value _. Having the ADT, it is possible to describe the dynamic components of a CO-OPN specification: the classes.

2.5 Class Modules

In this subsection we will show more detail on the classes that compose the DrinksDispenser system, and using this example explain the main elements of a CO-OPN model.

CO-OPN classes are described by means of modular algebraic Petri nets with particular parameterized external transitions which are the *methods* of the class. The behavior of transitions are defined by so-called *behavioral axioms*, similar to the axioms in an ADT. A method call is achieved by synchronizing external transitions, according to the fusion of transitions technique. The axioms have the following shape:

```
Cond => eventname With synchro : pre -> post
```

in which the terms have the following meaning:

- Cond is a set of equational conditions, similar to a guard;
- eventname is the name of a method with the algebraic term parameters;
- synchro is the synchronization expression defining the policy of transactional interaction of this event with other events, the dot notation is used to express events of specific objects and the synchronization operators are sequence, simultaneity and non-determinism.
- Pre and Post are the usual Petri net flow relation determining what it is consumed and what it is produced in the object state places.

CO-OPN provides tools for the management of graphical and textual representations.

In figure 6 the DrinksContainer class provides methods to modify, in different ways, the state representation of the drink container. For instance, the method addDrink is fireable if a init was performed assigning to the drink container the kind of drinks that can be kept in this container (place kind).

```
ADT Drink;
Interface
  Sort    drink;
  Generators
    Ice Tea , Soda , Beer , Whisky : -> drink;
End Drink;

ADT Coins;
Interface
  Use  Money;
  Sort coin;
  Generators
    5c ,10c ,20c ,50c : -> coin;
    1f ,2f ,5f : -> coin;
  Operation
    value _ : coin -> money;
Body
  Axioms
   (value 5c) = 5;  (value 10c) = 10;
   (value 20c) = 20;
   (value 50c) = ((2*(value 20c))+(value 10c));
   (value 1f) = (2*(value 50c));
   (value 2f) = (2*(value 1f));
End Coins;
```

Figure 5: The Drink and Coin ADTs

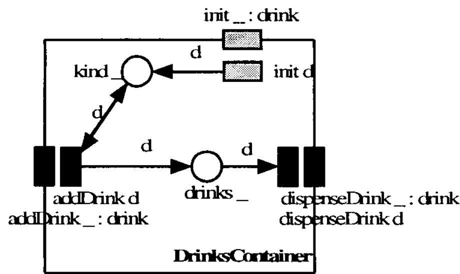

Figure 6: The Drinks Container class graphical description

A corresponding class `PhysicalDrinksContainers` (figure 7) is used to model the physical components of the dispenser. In our example the main difference of this class with respect to the controller class is the introduction of a limit in the number of loadable drinks. This will constrain the behavior of the whole model to a finite number of states. We consider that it is a natural way to take into account the physical limitation without over constraining the controller model.

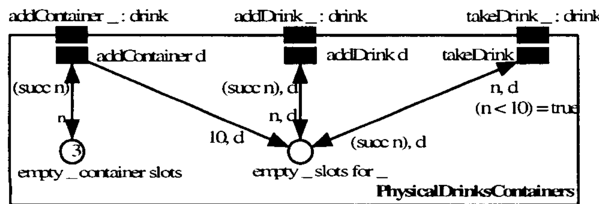

Figure 7: The PhysicalDrinksContainer class graphical description

In figure 8, the reader can see the textual description of the `centralUnit` Petri net. The axiom `ax1` illustrates complex properties that can be abstractly modelled using our formalism. The variable `c` represents a reference to container objects and is used to select one of the created containers. In `ax1` axiom the selected container will be the one that has the required kind of drink given by the variable `d`. The selection is made by the method that succeeds only when the kinds of drink match. As CO-OPN has logic based semantics, unbounded variable `c` (i.e. a variable that can be assigned to any of the tokens found in `container` place) is constrained by the firability of `dispenseDrink d` method of `DrinksContainer`. It means that `c` will be the object on which the method is firable. This example also illustrates the transactional semantics of CO-OPN (all-or nothing policy of synchronization) expressed in the `(this.takeMoney p)..(c.dispenseDrink d)` synchronization that can succeed if and only if both methods sequentially succeed.

```
Class CentralUnit;
Interface
  Use Drink, Money;
  Type  centralUnit;
  Gate   takeMoney _ : money;
  Methods
   distribute _ : drink;
   addContainer _ price _ : drink money;
   addDrink _ : drink;
Body
  Use DrinksContainer;
  Place
   container _ price _ : dc money;
  Axioms
   ax1:this = Self =>  distribute d With
     (this.takeMoney p)..(c.dispenseDrink d)::
     container c price p -> container c price p;
   ax2:addContainer d price p With
      c.init d::  -> container c price p;
   ax3:addDrink d With c . addDrink d::
     container c price p -> container c price p;
  Where
   this : centralUnit;    p : money;
   c : dc;  d : drink;
End CentralUnit;
```

Figure 8: The CentralUnit class textual description

3 Translation of CO-OPN to programming languages

The code generation process takes a CO-OPN specification as a parameter and produces a set of Java classes. The object structure of a CO-OPN specification is preserved by the generated code. One of our primary goals was to find a "natural" mapping between CO-OPN and Java. In such a mapping of standard CO-OPN features, constructs like methods or gates are associated to standard Java features, methods or events respectively. As a result, the interface part of a generated Java component is similar to the interface of the corresponding CO-OPN component, and it is also easy to understand/use by a human programmer or development tool. Some powerful aspects of CO-OPN, such as atomic concurrent synchronizations or non-determinism, do not have a direct equivalent in Java, consequently they are non-trivial to implement. These aspects are, as much as possible, hidden in private parts of the generated code.

We will continue by presenting the list of main problems solved in the code generator.

- Implementation of ADT using rewrite system [8] as explained in [7]. The structure of the generated code

allows one to replace parts of it by user code.

- Implementation of Classes and Contexts following the JavaBean structure [10] that provide a unified view of methods and gates. Nevertheless, additional features such as transactional support and non-determinism are managed by generated code.

Now we will give more insight in the complex process of generating Java classes and list the main concepts that are used:

- Transaction Support. The operational interpretation of CO-OPN synchronizations is based on the nested transactions principle [2]. Specific algorithms and data structures implement commit and abort operations.
- Non-determinism. Methods of CO-OPN classes may be non-deterministic in data and control dimensions. Data non-determinism occurs when a precondition takes values from places. It is possible that many different values match the precondition requirements. The choice from matching values is non-deterministic. Control non-determinism occurs when more that one method's axiom can apply in the given system state. Non-determinism is implemented in prolog-like fashion, nested transactions are used to undo changes of system state and code rewriting to implement the prolog "redo" primitive.
- Resource Sharing. In CO-OPN, concurrent synchronizations can share the same resources. A specific data structure that collaborates with the transaction mechanism has been devised to implement the rules of resource sharing.
- Concurrent Synchronizations. CO-OPN defines three kinds of concurrent synchronizations: sequence, simultaneity and non-determinism. The structure of the nested transaction tree follows the structure of the synchronization tree.

4 How to use the models

As explained before the purpose of modelling is not only to produce the controller, but also to be able to study the behavior of the machine that will be produced.

Now we will present two possibilities of using the code generator on the CO-OPN models.

First one is the validation of controller in "real" environment. The goal of validation is to insure that the controller can work with real systems. Clearly, the best way to test the software is by installing the controller program on the real drink distributor. Our less expensive approach is to use LEGO RCX [14] or similar system. In this approach a simplified hardware model of DDM is built with LEGO. The generated program is installed on a PC connected with the LEGO RCX computer via an infrared port. To avoid limitations of RCX system, a part of the interface of DDM runs on the PC. The resulting system can be used for demonstration purposes and to show its feasibility.

The second possibility is the use of code generation techniques to observe and verify properties of the whole DDM system without building it. As explained before, the DDM model contains a part that describes the constraints and the limitations introduced by the physical components. Instead of generating only a controller, we generate the code for the whole model for simulation purposes. This code can be executed to verify useful properties of the model. A tool (figure 9) exists that enables the user to execute CO-OPN synchronizations on generated code and watch for execution results.

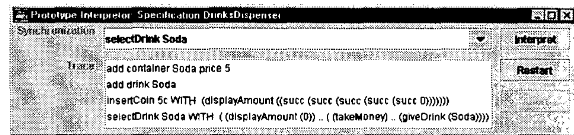

Figure 9: "Interpretor" tool

An example of a useful property for DDM could be: "If there is enough money in MB and enough drinks then the `selectDrink` request must always succeed". This property is then expressed as CO-OPN synchronization and submitted to generated code. The result can then be observed and eventual wrong behavior detected. As the state space of the DDM model is finite (because of limitations introduced by physical model) a quasi-exhaustive test of such a property is also possible.

5 Future Work and Conclusions

In this paper, we presented an attractive methodology for modelling embedded systems. Embedded software is modelled together with controlled equipment. In our opinion this is a technique that can be used for co-design.

We also propose a technique for automatically transforming a formal specification of a system into executable code. This technique was validated on a number of examples, including the drinks dispenser case study. We plan to continue the development of generation techniques to better suit the purposes of embedded software.

Generated code is also used for validation and verification of the model. We intend to develop this research direction in order to integrate test techniques.

References

[1] Didier Buchs and Nicolas Guelfi, ''A Formal Specification Framework for Object-Oriented Distributed Systems," IEEE TSE, vol. 26, no. 7, July 2000, pp. 635-652.

[2] G. Weikum, "Principles and Realization Strategies of Multilevel Transaction Management", ACM Transactions of Database Systems, Vol 16, No 1, pp 132-180, 1991.

[3] Olivier Biberstein, Didier Buchs, and Nicolas Guelfi; Object-oriented nets with algebraic specifications: The CO-OPN/2 formalism. In G. Agha, F. De Cindio and G. Rozenberg, editors, Advances in Petri Nets on Object-Orientation, volume to appear in Lecture Notes in Computer Science. Springer-Verlag, LNCS 2001, pp. 70-127.

[4] Didier Buchs and Mathieu Buffo. Rapid prototyping of formally modelled distributed systems. In Frances M. Titsworth, editor, *Proc. of the Tenth International Workshop on Rapid System Prototyping RSP'99*. IEEE, june 1999.

[5] Mathieu Buffo. Experiences in coordination programming. In *Proceedings of the workshops of DEXA'98 (International Conference on Database and Expert Systems Applications)*. IEEE Computer Society, aug 1998.

[6] Christine Choppy and Stéphane Kaplan. Mixing abstract and concrete modules: Specification, development and prototyping. In *12th International Conference on Software Engineering*, pages 173–184, Nice, March 1990.

[7] Didier Buchs and Jarle Hulaas. Evolutive prototyping of heterogeneous distributed systems using hierarchical algebraic Petri nets. In *Proceedings of the Int. Conf. on Systems, Man and Cybernetics*, Beijing, China, October 1996. IEEE.

[8] Ph. Schnoebelen. Refined compilation of pattern-matching for functionnal languages. *Science of Computer Programming*, pages 11:133-159, 1988.

[9] Olivier Biberstein and Didier Buchs. Structured algebraic nets with object-orientation. In *Proc.of the first int. workshop on "Object-Oriented Programming and Models of Concurrency" within the 16th Int. Conf. on Application and Theory of Petri Nets*, Torino, Italy, June 26-30 1995.

[10] Sun Microsystems: JavaBeans specification Version 1.01 (July, 1997).

[11] Mathieu Buffo and Didier Buchs. A coordination model for distributed object systems. In *Proc. of the Second International Conference on Coordination Models and Languages COORDINATION'97*, volume 1282 of *Lecture Notes in Computer Science*, pages 410–413. Springer Verlag, 1997.

[12] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs*. MIT Press, Cambridge and London, 1990.

[13] Jeff Kramer, Jeff Magee, Morris Sloman, and Naranker Dulay. Configuring object-based distributed programs in rex. *IEEE Software Engineering Journal*, 7(2):139–149, 1992.

[14] Lego Web site, http:\\mindstorms.lego.com