

Composing DisCo Specifications Using Generic Real-Time Events – A Mobile Robot Case Study*

Mika Katara

Tampere University of Technology, Software Systems Laboratory
P.O. Box 553, FIN-33101 Tampere, Finland
mika.katara@cs.tut.fi

Abstract Methods used to specify real-time control software should enable the expression of functional, control and real-time requirements. They should enable multi-disciplinary system development and promote reuse of specifications. This paper describes a specification of a real-time control software developed using the DisCo method. DisCo is an object-oriented action-based method with precise semantics in logic. The specification is layered and partly reusable. It consists of functional, control and real-time parts. The real-time part includes layers which specify generic periodic and aperiodic events. The control part specifies the control algorithms, and the functional part the rest of the system. The three parts are specified using stepwise refinements and combined in a simple way. Although the specification presented is quite small, the techniques used are applicable when specifying larger systems with complex real-time behavior.

1 Introduction

Real-time systems frequently include control of some physical devices. Therefore their design demands knowledge of both continuous and discrete worlds, making it a demanding task. As systems grow in complexity, the role of the control software specification is emphasized in the design process.

Methods for specifying real-time control software should enable the expression of functional, control and real-time requirements. They should enable multi-disciplinary system development [3] and facilitate software maintenance. Furthermore, methods should promote reuse at the specification level. Jacky used Z (see, e.g. [15]) to specify a cyclotron control system as reusable frameworks [6].

DisCo¹ [7] is an object-oriented method for specifying and reasoning about reactive and distributed systems. The method incorporates a specification language, a methodology for developing specifications using the language, and tool support for the methodology [14,8]. DisCo is designed for software engineers rather than theoreticians, and thus the DisCo language resembles more a programming language than mathematical formulas. However, every DisCo specification has a precise interpretation in Temporal Logic of Actions (TLA) [11], which enables rigorous reasoning.

* In Jaan Penjam, editor, *Software Technology, Proceedings of the Fenno-Ugric Symposium FUSST'99*, pages 75–86, Sagadi, Estonia, August 1999. Institute of Cybernetics at Tallinn Technical University (Technical Report CS 104/99).

¹ Acronym for *Distributed Co-operation*.

In this paper, a specification of a mobile robot control software is presented. The specification consists of functional, control and real-time parts. The parts are given separately but are combined as the level of abstraction is lowered using stepwise refinements. The real-time part includes layers which define generic periodic and aperiodic events. The specification was implemented in the C language.

The generic real-time events were first introduced in [10], which describes the methodology in conjunction with closed world modelling, object-orientation and distribution, and where the control software specification served as an example. In this paper the generic aperiodic event is generalized even a bit more (quantified roles) and also some light is shed on the implementation.

This paper is structured as follows. In Section 2 the DisCo method is introduced. The method is applied in Section 3 where the case study is presented. Conclusions are reported in Section 4 together with the description of a future work.

2 Introduction to DisCo

DisCo is based on the joint action theory developed by Back and Kurki-Suonio [1,2], in which objects communicate in atomic *actions*. The joint action paradigm has been found suitable for specifying and reasoning about reactive and concurrent systems. Instead of processes we are interested in actions that may have one or more *participants*. Notations for processes are not needed, because they can be modelled by objects which participate in actions.

A DisCo specification consists of *layers*. Each layer gives an abstract view of the system and has an operational interpretation. All variables in a layer are encapsulated in objects. DisCo also has *classes* and *inheritance*, familiar from other object-oriented languages. Methods are replaced by actions, having neither callers nor callees.

In an action each participant is assigned a *role*. Objects are treated as data structures capable of participating in actions in certain roles. Actions may also have *parameters* which do not refer to any objects. Parameters, which have nondeterministic values, can be used to introduce nondeterminism.

In an action definition the roles and the classes of participating objects are given. A Boolean expression, called the *guard*, and the *body* of the action are also given. The body consists of assignment and conditional statements. If the role name is given in braces, the role is said to be *quantified*. It means that a nondeterministic (possibly empty) set of objects that satisfy the guard can participate in the role.

Actions in DisCo are atomic and are executed sequentially. An action is said to be *enabled* if there exist potential participants so that the guard is true. If more than one action is enabled the one to be executed is selected nondeterministically. The execution of an action means executing the statements given in its body.

As an example, consider a simple layer given below which models sending and delivering messages holding integer values. The layer defines classes Message and Receiver and actions Sending and Delivery. The parameter *i* of action Sending, the value of which is copied to the message participating as *m*, models the value to be sent. In action Delivery the value within participant *m* is delivered to all receivers participating in the quantified role *r*:

```

layer Messages is
  class Message is
    i: integer;
  end Message;

  class Receiver is
    i: integer;
  end Receiver;

  action Sending (m: Message; i: integer) is
  when true do
    m.i := i;
  end Sending;

  action Delivery ({r}: Receiver; m: Message) is
  when true do
    r.i := m.i;
  end Delivery;
end Messages;

```

In DisCo, specifications are refined by importing one or more layers into a new layer and applying the *superposition* principle. Classes can be extended with new components, and totally new classes can be added. New actions can be given, and new participants and parameters can be added to the old actions. Also, the guards can be strengthened and action bodies extended by giving new statements. However, actions are not allowed to include assignments to the variables introduced in imported layers. This restriction is necessary to ensure the preservation of safety properties.

Actions can be *synchronized* and *specialized*. Synchronizing means that two or more actions are combined to execute in parallel. Synchronizing two actions does not remove the non-synchronized versions of actions by default. Specializing means introducing a new action that is specialized for a subclass, i.e., it is enabled only for objects of the subclass. The guards of the other versions of the action are implicitly strengthened so that they are no longer enabled for the subclass.

Because our execution model does not guarantee that all enabled actions are finally executed, fairness requirements have to be given to ensure liveness. Prefixing the name of a (possibly quantified) role with an asterisk indicates that if an action is infinitely often enabled so that the same object can participate in the same prefixed role, then the action has to be executed infinitely often with this object in the prefixed role. In fact, explicit fairness requirements are the biggest difference between the execution models of DisCo and UNITY [5].

Consider as an example a simple refinement of layer Messages given below. A subclass FirstClassMessage of Message is introduced and Delivery is specialized for it. Moreover, fairness with respect to participants in the quantified role *r* is required (ellipses refer to the parts given in the imported action):

```

layer FirstClassMessages is
  import Messages;

```

```

class FirstClassMessage is new Message;

specialized FirstClassDelivery ({*r}: Receiver; fcm: FirstClassMessage)
of Delivery({r},fcm) is
when ... do
...
end FirstClassDelivery;
end FirstClassMessages;

```

In layer `FirstClassMessages` there are two versions of `Delivery`, one for `Messages` which are not `FirstClassMessages` and another, named `FirstClassDelivery`, for `FirstClassMessages`.

Real time is modelled in `DisCo` as follows. Actions are assumed to be instantaneous. Variable Ω is used to represent the current time after the execution of an action, and variable Δ a multiset containing deadlines for actions. Variable Ω is initialized as 0.0 and Δ as empty.

An implicit parameter τ representing the time when an action is executed is added to each action. Also, all guards are strengthened by the conjunct

$$\Omega \leq \tau \leq \min(\Delta).$$

Moreover, an assignment $\Omega := \tau$ is added to all action bodies. In the presence of real time, liveness still remains the only execution force. In the sequel, τ is denoted by keyword `now` and operations on Δ by operator `@`.

It should be noted that initializing the global set of deadlines Δ as empty is not absolutely necessary. There might initially exist some deadlines as long as they are removed correctly.

3 Case Study

The mobile robot is a small microcontroller-based car. The objective is to keep the car on a track marked by optical tape. From the viewpoint of the control software the system has two inputs and two outputs. The inputs are readings from an A/D converter connected to six infra-red sensors, and from an odometer which receives a pulse every 0.75 cm. There is also a switch, which is used to start the car and to stop it.

The outputs are, from the viewpoint of the software, two servo motors controlling the steering and the movement. The servos are driven by PWM (Pulse Width Modulation) signals. Pulses for the servos have to be generated every 20 ms.

The functional part of the specification consists of two consecutive layers, which are named `Basic_Actions` and `Drive_States`. Layer `Control_Algorithms` forms the control part of the specification. The real-time part is formed by four layers, `Periodic`, `Aperiodic`, `Periodic_Instance` and `Aperiodic_Instance`. The three parts are combined in layer `Combined_Specification`. The import scheme is illustrated in Figure 1.

3.1 Functional Properties

Layer `Basic_Actions` contains the model of the environment and the basic functionality of the system. There are two classes and three actions. Class `Data` holds internal

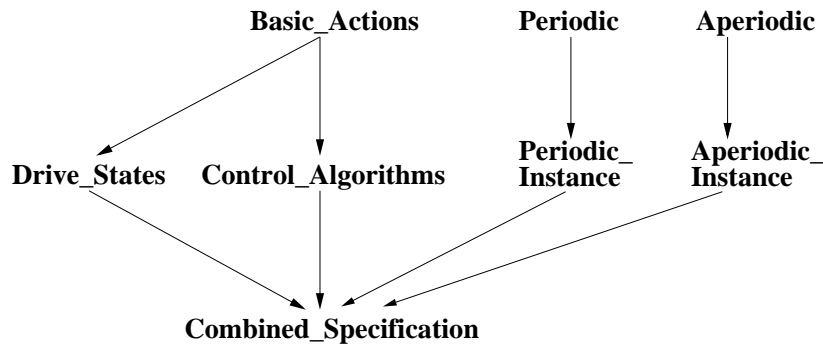


Figure1. Import scheme of the specification

variables and class Output variables that model the outputs. The variables `r_dist` and `r_tape` of type `real` represent the distance covered between the last two readings of the odometer and the location of the car relative to the tape, respectively. Variables `c_engine` and `c_steer` model the current lengths of the servo pulses. If both equal zero, the car is stationary with its wheels straight. In a class definition, the number of instances is indicated by placing the number in parentheses after a class name. The classes are shown below:

```

class Data (1) is
    r_dist: real := 0.0; r_tape: real := 0.0;
end Data;
  
```

```

class Output (1) is
    c_engine: real := 0.0; c_steer: real := 0.0;
end Output;
  
```

Action `Clear` clears all the variables given in this layer. Action `Read`, which has a participant of class `Data`, models the reading of the odometer and the A/D converter. The actual new readings are modelled by two parameters `r_x` and `r_y`. In action `Control`, parameters `c_x` and `c_y` are assigned to the variables `c_engine` and `c_steer`, respectively. The parameters are used to model the new values given by the control algorithm. The actions are given below:

```

action Clear (D: Data; O: Output) is
when true do
    D.r_dist := 0.0; D.r_tape := 0.0;
    O.c_engine := 0.0; O.c_steer := 0.0;
end Clear;
  
```

```

action Read (r_x, r_y: real; D: Data) is
when true do
    D.r_dist := r_x; D.r_tape := r_y;
end Read;
  
```

```

action Control (c_x, c_y: real; O: Output) is
when true do
    O.c_engine := c_x; O.c_steer := c_y;
end Control;

```

Because the guards of all three actions are identically true, the actions are continually enabled. The behavior of the system consists of clearing the variables, reading the inputs and writing the outputs. The order in which these actions are executed is non-deterministic.

Layer Drive_States introduces the start/stop switch and specifies the order in which the actions are executed.

Class **Data** is extended to hold a state machine **d_state**, which indicates the actions that are allowed to be executed. The state machine has states **start**, **read** and **control**, the first of which is the default state.

The switch is modelled by variable **switch**, which has two states, **on** and **off**. The state of the switch is changed in action **Toggle**. When the switch is **on** in state **start**, action **Start** is enabled. It changes the state to **read**.

Fairness is required with respect to the participant of class **Data** in every action except **Toggle**, which is executed by the environment. The extensions of **Data** and the new actions are shown below:

```

extend Data by
    d_state: (start, read, control);
    switch: (off, on);
end Data;

```

```

action Toggle (D: Data) is
when true do
    if D.switch'off then
        D.switch -> on();
    else
        D.switch -> off();
    end if;
end Toggle;

```

```

action Start (*D: Data) is
when D.switch'on and D.d_state'start do
    D.d_state -> read();
end Start;

```

The car is fully operational when the switch is **on** in states **read** and **control**. Likewise, actions **Read** and **Control** are refined so that they are enabled correspondingly. Furthermore, by addition of state transition statements **D.d_state -> control()** and **D.d_state -> read()** to **Read** and **Control**, respectively, they are executed by turns. The refined action **Read** is given below:

```

refined Read (r_x, r_y : real; *D: Data) is
    when ... D.switch'on and D.d_state'read do
        D.d_state -> control();
    ...
end Read;

```

Furthermore, action Clear is refined to change the state back to start when the switch is turned off. In this case it implicitly stops the engine and straightens the wheels by clearing all the variables. The refined action is named Stop.

3.2 Control Properties

Layer Control Algorithms contains the structures needed to control the movement and the steering. The layer imports layer Basic_Actions and defines ten constants, extends class Data, introduces two functions and refines all three actions given in Basic_Actions. The constants and the variables are added due to the control algorithms. Variable e_state represents the state of the engine. The three states power_up, moves and normal are needed since, because of friction, it is necessary to power up until movement is sensed and after that power down slightly to prevent slipping.

The P and PID algorithms are used to compute new values for the engine and steering, respectively. Function PID is shown below (s_P, s_I and s_D are constants) :

```

function PID(D: Data) : real is
    return -s_P*D.r_tape + s_I*D.r_tape_ma + s_D*(D.r_tape_old - D.r_tape);
end PID;

```

Action Read is refined to include the statements needed by the control algorithm (comments begin with double hyphens):

```

refined Read (r_x, r_y: real; D: Data) is
when ... do
    if (r_x = 0.0) and (D.r_dist = 0.0) then
        D.e_state -> power_up();
    elseif (r_x > 0.0) and (D.r_dist = 0.0) then
        D.e_state -> moves();
    else
        D.e_state -> normal();
    end if;

    D.r_tape_ma := ((n - 1)*D.r_tape_ma - D.r_tape)/n;
    D.r_tape_old := D.r_tape;
    ...
end Read;

```

In the guard of action Control, parameters c_x and c_y are bound to the return values of functions P and PID, respectively.

If the car has lost the track, it should stop. This is the situation if the absolute value of r_tape is greater than limit, which is treated as a special case in the guard of action Control. The refined action Control is shown below:

```

refined Control (c_x, c_y: real; O: Output; D: Data) of Control(c_x,c_y,O) is
when ... (c_x = if((abs D.r_tape) > limit) then 0.0 else P(D,O) end if)
    and c_y = PID(D) do
    ...
end Control;

```

Furthermore, action Clear is refined to clear variables introduced in this layer.

3.3 Real-Time Properties

Layer Periodic specifies a generic periodic event, which takes place exactly at given intervals. The layer consists of one class and one action. The class `Periodic_data` contains variables `d` of type `real` and `t` of type *time*, which is a synonym type of `real`. The former is used to store the period, and the latter to store the deadline:

```

class Periodic_data is
    d: real;
    t: time;
end Periodic_data;

```

Action `P_Timeout` has one participant of class `Periodic_data`. The action is enabled when `now` is greater than or equal to the current deadline, i.e., when enough time has passed. The default value for variables of type `time` is 0.0, so the action is enabled initially. The action first removes the possible deadline stored in `pd.t` from the set of deadlines and then assigns a new one at `now + pd.d`. The deadline is set using the binary version of operator `@`. It adds the deadline to the global set of deadlines, and stores it also to the left operand. The deadline is removed from the global set of deadlines by applying unary `@` to the variable storing the deadline:

```

action P_Timeout (*pd: Periodic_data) is
when pd.t <= now do
    pd.t @;                - - remove the old deadline
    pd.t @ pd.d;          - - set a new one
end P_Timeout;

```

The guard of the action ensures that it cannot be executed too early and the deadline that it cannot be executed too late. This implies that the interval between two executions of `P_Timeout` for one object participating as `pd` equals `pd.d`.

The generic periodic event is instantiated by inheriting class `Periodic_data`. The class itself is abstract. Its objects should belong to singleton subclasses, i.e., subclasses in which there is only one object. Subclasses should be singletons, because they are used to identify different real-time requirements.

It should be noted that fairness with respect to participant `pd` is required. Otherwise it would be possible that the time stops and the action is never executed.

Layer Aperiodic specifies a generic aperiodic event similarly to the periodic event. The layer has one class and two actions. The class `Aperiodic_data` is similar to `Periodic_data`. Actions `AP_SetTimeout` and `AP_Timeout` are used to set and remove deadlines, respectively. Action `AP_SetTimeout` has a quantified role `apd` for class `Aperiodic_data`. The action sets deadlines `now + apd.d` for every object participating in role `apd`:

```
action AP_SetTimeout ({*apd}: Aperiodic_data) is  
when true do  
    apd.t @ apd.d;  
end AP_SetTimeout;
```

Action `AP_Timeout` has also a quantified role of class `Aperiodic_data`. When enough time has elapsed, the action becomes enabled. When executed, it removes the deadlines stored in variables `apd.t` of objects participating in role `apd`:

```
action AP_Timeout ({*apd}: Aperiodic_data) is  
when apd.t = now do  
    apd.t @;  
end AP_Timeout;
```

Like in the periodic case, the guard of the action `AP_Timeout` and the deadlines ensure that the action is executed exactly when deadline intervals have elapsed. There could also be a similar action with the guard expression replaced by `true`. It could be executed before the deadlines or exactly at them.

Again, the generic aperiodic event is instantiated by inheriting class `Aperiodic_data`. The class itself is abstract. Objects of `Aperiodic_data` should belong to singleton subclasses. However, it should be noted that although these actions allow us to set and remove many deadlines simultaneously, it does not imply that the deadlines set simultaneously should also be removed simultaneously.

Layers Periodic_Instance and Aperiodic_Instance are used to instantiate the generic events. Layer `Periodic_Instance` imports layer `Periodic` and `Aperiodic_Instance` imports `Aperiodic`.

The generic periodic event is instantiated by introducing a singleton subclass `P1` of class `Periodic_data` and specializing action `P1_Timeout` for the subclass:

```
class P1 (1) is new Periodic_data;  
  
specialized P1_Timeout (*p : P1)  
of P_Timeout(p) is  
when ... do  
    ...  
end P_Timeout;
```

Similarly, the generic aperiodic event is instantiated by introducing a singleton subclass `AP1` of class `Aperiodic_data` and specializing `AP1_SetTimeout` and `AP1_Timeout` for the subclass.

3.4 Combining Specification Layers

Layer Combined Specification combines the functional, control and real-time parts of the specification. It imports layers `Drive_States`, `Control_Algorithms`, `Periodic_Instance` and `Aperiodic_Instance` (see Figure 1). The parallel refinements of actions and extension of classes are combined automatically, e.g. action `Read` has all the characteristics introduced in layers `Drive_States` and `Control_Algorithms`. Furthermore, because `Clear` and `Stop` are refinements of the same action they are combined automatically and the combination is named `Clear&Stop`.

Actions `Read` and `Control` were refined in such a way that the logic ensures that they are executed by turns. The objective is to make `Read` and `Control` execute periodically. This is achieved by forcing `Read` to execute periodically and `Control` directly after it.

Actions `AP1_SetTimeout` and `P1_Timeout` are synchronized with action `Read` and the combined action is named `Periodic_Read`. Every time the action is executed the effect is the same as if all three actions were executed in parallel so that the deadlines are stored in the variables of objects of classes `P1` and `AP1`. Synchronization is done in the import part of the layer using include clause:

```
include AP1_SetTimeout&P1_Timeout&Read(AP1, P1, real, real, Data)
as Periodic_Read(AP1, P1, real, real, Data);
```

Every time `Periodic_Read` is executed, two deadlines are set. One is for the action itself and the other is for `Control`. Because `Control` should remove the deadline set for it, the action is synchronized with `AP1_Timeout`:

```
include AP1_Timeout&Control(AP1, real, real, Output, Data)
as Aperiodic_Control(AP1, real, real, Output, Data);
```

Moreover, if the switch is turned off between actions `Read` and `Control`, action `Clear&Stop` should remove the deadline set for `Control`. Thus, it is synchronized with `AP1_Timeout`:

```
include AP1_Timeout&Clear&Stop(AP1, Data, Output)
as Stop(AP1, Data, Output);
```

The guard of the non-synchronized version of `P1_Timeout` is strengthened so that it is not enabled while `Periodic_Read` or `Aperiodic_Control` are enabled. The resulting action is named `Mere_P1_Timeout`. Intuitively this means that if the car is in state `start` nothing is done when the periodic event happens:

```
refined Mere_P1_Timeout (*p : P1; D: Data)
of P1_Timeout(p) is
when ... D.d_state'start do
  ...
end Mere_P1_Timeout;
```

The non-synchronized versions of actions `Read`, `Control`, `AP1_Timeout`, `AP1_SetTimeout`, and `Clear&Stop` should not be executed, so their guards are strengthened to false. Actions `Toggle` and `Start` are taken as such.

3.5 Implementation

The specification was implemented in the C language. The implementation was straightforward. Some parts of the specification were just copy-pasted and edited to get C code. With comments, there are about 400 lines of code in the implementation.

The architecture of the program is as follows. Besides main function there are six other functions and an interrupt handler. The functions consists of an initialization function, two functions which read the odometer and the A/D converter, two functions which compute new values to drive the servo motors (using P and PID algorithms), and a function which clears the variables.

The control flows as follows. After initialization the program waits for the user to switch on the car. Subsequently it goes into an infinite loop to wait for interrupts. When an interrupt occurs, the interrupt handler calls functions which read values from the odometer and the A/D converter. Consecutively, the new values to drive the servos are computed. If the switch is off when the interrupt occurs, the handler only calls the function which clears the variables.

Some changes had to be made to the specification in the implementation phase because some important details of the underlying architecture were not discovered early enough. However, specifying a system is often an iterative process.

4 Conclusions and Future Work

Different aspects of the specification can be encapsulated in separate layers [13,10]. The specification presented in this paper gives an example of using such *logical layers*. The most important characteristics of the specification are:

- the specification is composed of functional, control and real-time parts,
- generic events were easy to specify,
- generic events can be instantiated using object-oriented inheritance, and
- combination of the three parts was simple.

A layered DisCo specification gives a facile basis for maintenance. It serves as a document on what has been done. New features can be added by giving new layers. Because different layers are just abstract views of the same system, multi-disciplinary system development is possible. For example, a control engineer can specify the control related parts, which are later combined with the other parts of the specification. Preservation of safety properties seems an essential feature of the method in this case.

Methods for real-time specification should contain abstractions which directly relate to common real-time activities [4]. The generic real-time events introduced can be used to give complex real-time requirements. Furthermore, other real-time layers can be specified in a similar manner. DisCo is well suited for specifying such *patterns* [12].

In the specification presented the physical environment of the car was modelled using nondeterminism. Exact modelling of the environment would have involved continuous functions of time to describe how the values of parameters r_x and r_y change when c_x and c_y are changed. Without rigorous modelling of the environment it is not possible to reason about all the properties that might be of interest. It seems evident

that the DisCo method should be developed towards *hybrid modelling*, involving both discrete and continuous behavior. Moreover, as indicated in [9], this is theoretically a natural way to proceed.

Acknowledgments

The research presented in this paper was partly funded by the Academy of Finland (project 21490). The author would also like to thank all the other members of the DisCo project, for their creative ideas, and Anssi Toivo for constructing the car.

References

1. Back, R.J.R., Kurki-Suonio, R., Distributed Cooperation with Action Systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, Oct. 1988.
2. Back, R.J.R., Kurki-Suonio, R., Decentralization of Process Nets with Centralized Control. *Distributed Computing* 3 (1989), pp. 73–87. An earlier version in *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Montreal, Canada, Aug. 1983, pp. 131–142.
3. Brink, K., van Katwijk, J., Toetenel, W.J., Applying Formal Software Requirements Specification in the Development of Control Applications. In *Proceedings of the first annual conference of the Advanced School for Computing and Imaging*, Heijen, The Netherlands, May 1995, pp. 11–17.
4. Burns, A., Wellings, A.J., HRT-HOOD: A Structured Design Method for Hard Real-Time Systems. *Real-Time Systems*, 6(1):73–114, Jan. 1994.
5. Chandy, K.M., Misra, J., *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
6. Jacky, J., Specifying a Safety-Critical Control System in Z. *IEEE Transactions on Software Engineering*, 21(2):99–106, Feb. 1995.
7. Järvinen, H.-M., Kurki-Suonio, R., Sakkinen, M., Systä, K., Object-Oriented Specification of Reactive Systems. In *Proceedings of the 12th International Conference on Software Engineering*, Nice, France, Mar. 1990, pp. 63–71.
8. Kellomäki, P., Verification of Reactive Systems Using DisCo and PVS. In *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, LNCS 1313, Springer Verlag 1997, pp. 589–604.
9. Kurki-Suonio, R., Hybrid models with fairness and distributed clocks. In *Hybrid Systems*, LNCS 736, Springer-Verlag 1993, pp. 103–120.
10. Kurki-Suonio, R., Katara, M., Logical Layers in Specifications with Distributed Objects and Real Time. To appear in *Computer Systems Science & Engineering*.
11. Lamport, L., The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
12. Mikkonen, T., Formalizing design patterns. In *Proceedings of the 20th International Conference on Software Engineering*, Kyoto, Japan, April 1998, pp. 115–124.
13. Mikkonen, T., *Abstractions and Logical Layers in Specifications of Reactive Systems*, Doctoral dissertation, Tampere University of Technology, 1999.
14. Systä, K., A Graphical Tool for Specification of Reactive Systems. In *Proceedings of the Euromicro'91 Workshop on Real-Time Systems*, Paris, France, June 1991, pp. 12–19.
15. Wordsworth, J.B., *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering*. Addison-Wesley, 1992.