Design Approach for Real-Time Reactive Systems

Mika Katara*
Software Systems Laboratory
Tampere University of Technology
P.O. Box 553, 33101 Tampere, Finland
Mika.Katara@cs.tut.fi

Tommi Mikkonen
Nokia Telecommunications
P.O. Box 785, 33101 Tampere, Finland
Tommi.Mikkonen@nokia.com

Abstract

We propose a design approach for real-time reactive systems. The core of the approach is to use abstract real-time requirements or constraints arising from different views of a system. These requirements are specified separately in the early phases of the development. When conjoining such views, it is a design decision to partition real-time requirements given in the views into those needed for lower-level concepts. An explicit introduction of real-time requirements in a high-level specification, and their verifiable partitioning into implementable operations then provides justification for the final design. The approach is demonstrated with an example, where requirements related to telecommanding of a simple device are first formalized in an abstract fashion, and then partitioned into implementation-level operations.
Keywords: reactive systems, action systems, formal methods, real time, partitioning

1 Introduction

As real-time systems grow in complexity their development becomes increasingly difficult. This difficulty culminates in the development of highly dependable systems, whose development approaches should enable early considerations and justification of timeliness properties in order to support necessary design decisions.

In this paper, we address the specification of such systems with the DisCo method. DisCo [7] is a formal specification method for *reactive systems*, i.e., those that are in continuous interaction with their environments. The method incorporates a specification language [6], a methodology for developing specifications using the language [17], and tool support for the methodology [20, 8]. DisCo is based on the joint action theory by Back and Kurki-Suonio [1, 2], which enables the specification of collective behaviors at a high-level of abstraction in the beginning of the development. DisCo supports the refinement of the level of abstraction of interactions, whereas conventional methods restrain the designer to establish the interfaces between the implementation level components before fully understanding the interactions between them. This is achieved by relying on specifications that describe closed systems, i.e. systems together with their assumed environments. When combined with joint actions, the closed-system approach allows to specify the collective behavior of the components before their interfaces. For hardware-software codesigns, the closed-system paradigm and the possibility to fine-tune the level of abstraction between components provides essential support [18].

In DisCo, different *aspects* or *views* of a system

*Funded by the Tampere Graduate School in Information Science and Engineering (TISE).

are modeled as individual *logical layers*. This enables specification architectures that are orthogonal to component-based formalizations, as the unit of modularity does not reflect straightforward implementation mapping [19]. Layers can be refined using *superposition* [5], and independently given layers can be conjoined. It is also possible to give refinements that turn some parts of the specification into abstractions that need no explicit representation in an implementation. This requires proving an invariant that determines the relation of the abstraction and its implementation.

Each layer may also introduce real-time requirements or constraints that should be satisfied cooperatively by the components involved. Such requirements may need to be *partitioned* into lower level requirements when the specification is refined towards an implementation. In this paper we concentrate on two important timeliness requirements: *bounded response* and *minimal separation*, which give upper and lower bounds for delays between events, respectively. Other important real-time abstractions such as periodic and aperiodic events can be specified using them (see e.g. [10]). Verification and implementation fall outside the scope of this paper.

The rest of the paper is structured as follows. Section 2 introduces the DisCo method. In Section 3 the design approach is discussed and Section 4 provides an example. Finally, Section 5 concludes the paper.

2 Introduction to DisCo

This section gives a short introduction to the use of the DisCo method. Basic knowledge of formal specification is assumed.

2.1 Classes, Relations and Actions

DisCo specifications are composed in terms of *classes*, *relations*, and *actions*. Classes are patterns for *objects*, introduced in the format

$$\textbf{class}\ C = \{contents\},$$

where *contents* refer to variables that constitute the *local state* of objects in class C. Record types are defined similarly. Available primitive types are boolean, integer, real, and time, a synonym type of real.

Objects can be associated with one another with relations. Relations are defined for different classes in the format

$$\textbf{relation}\ m \cdot R \cdot n : C \times D,$$

where relation R associates an instance of class C with n instances of D, and an instance of D with m instances of C. Ranges are also allowed, and an asterisk is used as a shorthand for an arbitrary number of instances.

Actions are used to define how the *global state* of a system, constituted by the local states of all objects and established relations, evolves over time. Actions are atomic steps of execution, i.e., once started their executions are bound to be finished without outside interference. Each action defines a collaborative operation where the local states of the involved objects can be modified, and relations associated with the objects can be set and reset. Actions are given in the format

$$A(x : C; y : D) : g(x, y) \to b(x, y),$$

where x and y are *roles* for *participating* objects of classes C and D. Boolean expression $g(x, y)$ is an enabling condition referred to as a *guard*. The *body* of the action, $b(x, y)$, defines the state changes caused by the action. In an action body, unprimed and primed variable names can be used, referring to the values of variables before and after an execution, respectively. Variables that are not assigned to preserve their values. An action that alters no value of a variable in *V*, a set of variables defined in the objects of the specification, is called a *V-stuttering* step. Actions may also involve *parameters* which do not refer to any objects.

Actions are executed in an interleaving fashion. The action to be executed next is nondeterministically selected from those that are *enabled*, i.e., those whose enabling condition is true. Objects and the values of parameters are selected nondeterministically from those for which the action can be executed.

2.2 Temporal Logic of Actions

The formal basis of the DisCo language is in the *Temporal Logic of Actions, TLA* [13]. For the actual mapping the reader is referred to [6].

TLA is a linear time logic where attention is on infinite sequences of states called *behaviors* and their properties. An infinite number of program variables is assumed. In each state of a behavior every variable has a value. A pair of consecutive states is called a step. Actions are "step predicates" which are evaluated for steps. Unprimed and primed variable names are used to distinguish values of variables in the first and the second state of a step, respectively.

If all variables in *U* have the same values in both states of a step, the step is called a *U-stuttering step*,

$$Stut_U \triangleq \forall u \in U : u' = u.$$

Furthermore, stuttering extension of any action $\mathcal{A}$ (script letters are used to distinguish TLA actions from DisCo actions) is defined as

$$[\mathcal{A}]_U \triangleq \mathcal{A} \vee Stut_U,$$

and the non-stuttering version will be denoted by

$$\langle\mathcal{A}\rangle_U \triangleq \mathcal{A} \wedge \neg Stut_U.$$

When the set of variables is clear from the context, we will omit the subscripts.

Expressions are evaluated for behaviors. We use operators "$\Box$" (always),"$\Diamond$" (eventually), and "$\leadsto$" (leads to) that have the following relations:

$$\Diamond E \triangleq \neg\Box\neg E,$$
$$\langle\mathcal{A}_1\rangle \leadsto \langle\mathcal{A}_2\rangle \triangleq \Box(\langle\mathcal{A}_1\rangle \Rightarrow \Diamond\langle\mathcal{A}_2\rangle),$$

where E refers to an expression, and A_1 and A_2 are actions. Intuitively, $\langle\mathcal{A}_1\rangle \leadsto \langle\mathcal{A}_2\rangle$ requires that each step $\langle\mathcal{A}_1\rangle$ will always eventually be followed by a step $\langle\mathcal{A}_2\rangle$. The operator is transitive,

$$\langle\mathcal{A}_1\rangle \leadsto \langle\mathcal{A}_2\rangle \wedge \langle\mathcal{A}_2\rangle \leadsto \langle\mathcal{A}_3\rangle \Rightarrow \langle\mathcal{A}_1\rangle \leadsto \langle\mathcal{A}_3\rangle.$$

2.3 Layerwise Specification

DisCo specifications consist of *logical layers*. Such layers remind program slices [21] or projections used in verification [12]. In DisCo, however, layers are not used for decomposition of existing systems, but for introduction of new aspects or views of the specification.

When a new layer is incorporated in a specification, the layer may introduce new classes, relations, and actions, and extend classes and refine actions given in already existing layers. However, when actions are refined, they are required to logically imply the ones they have been derived from. Restriction to such refinements force the layers to apply superposition, a well-known technique first used in [5]. With such disjoint layers, stuttering actions of one layer may be essential for other layers. Due to allowing stuttering, all actions of the complete specification can be interpreted to be executed even when the behavior of the complete system is projected on one layer which is examined in isolation. This enables refinements that introduce more details to abstract behaviors.

With such restrictions, the characteristic property of a DisCo layer is that it never violates *safety properties* of other layers. Such properties consist of statements of the form "Something bad will never happen." *Liveness properties*, i.e., statements of the form "Something good will eventually happen," require additional proofs, because layers may strengthen enabling conditions of actions.

In practice, we give class extensions in the format

$$\textbf{class } C_2 = C_1 + \{extensions\},$$

where indices are used to refer to different layers. In this paper, layers are explicitly indicated only when they are not obvious from the context. Otherwise, the latest available version is assumed to be used. Actions are refined in the format

$$\begin{aligned} A_2(x : C; y : D; z : E) : \textbf{refines } A_1(x, y) \\ \wedge\, g_2(x, y, z) \\ \rightarrow b_2(x, y, z). \end{aligned}$$

If a new action is a refinement of several actions, it is required to logically imply all the base actions.

There is no notion of implicit liveness. Instead, operations that are required to take place must be indicated with *fairness requirements*. Such requirements can be attached to a role given for objects in actions by adding an asterisk to role names. The requirement then ensures that if an object could repeatedly take the role in an action execution, the action will be executed for this object infinitely often. This is a considerable methodological difference when compared to UNITY [3] where uniform fairness requirements are given with respect to all actions. Otherwise, the execution models of the two formalisms are very similar.

2.4 Timed Specifications

Real time can be incorporated in the above scheme in the following fashion (for more detailed discussion, see e.g. [10]). Each action is assumed to be executed instantaneously. A real-valued clock variable Ω is introduced to record time from the beginning of a behavior, and all actions are taken as refinements of action

$$\begin{aligned} Record\_time(\tau : \mathbb{R}) : \Omega \leq \tau \leq min(\Delta) \\ \rightarrow \Omega' = \tau. \end{aligned}$$

Parameter τ indicates the execution moment of the action, and Δ a multiset of *deadlines*, initialized as empty.

Minimal separation requirement between actions A and B can be enforced by strengthening the guard of action B by conjunct $\tau \geq \tau_A$ + d, where τ_A denotes the most recent execution moment of A. For bounded response requirements, deadlines are used. When a deadline $\tau + d$ is needed for some future action, a statement of the form $x' = \Delta\_on(d)$ is given in the action body to add this deadline to Δ and to store it in a variable x. The conjunct $\tau \leq min(\Delta)$ in all guards then prevents advancing Ω beyond this deadline, until some action removes the deadline with $\Delta\_\mathit{off}(x)$.

The real-time properties are associated with individual linear behaviors, so worst-case properties can be expressed, but issues related to average efficiency cannot. In the model, fairness still remains the only execution force. Actions are not executed because time passes, but the passing of time is noticed as a result of executing an action.

3 The Design Approach

The design process adopted in DisCo has been described in [15, 9]. In this context, we introduce the main aspects related to the modeling of real-time systems.

3.1 Views as Basis for Layers

The DisCo method enables modeling in a fashion where *aspects* or *views* of a system constitute specification modules. Each such module is given as a separate logical layer, potentially affecting several parties of the implementation. This allows

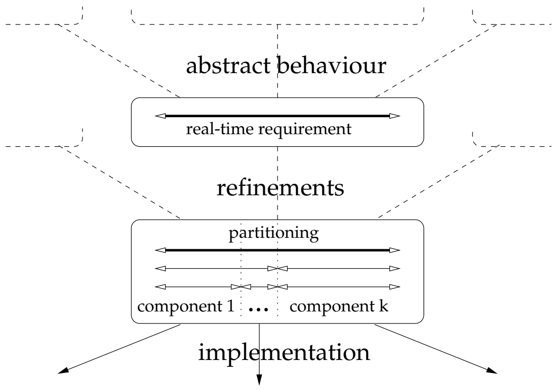

Figure 1: Top-down design process

specification architectures that are orthogonal to component-based formalizations, since specification modules need not reflect a straightforward implementation mapping [19]. This is in line with [4], where the need for something that extends beyond individual objects and ties them together is acknowledged. Moreover, rigorous reasoning with respect to behaviors can usually be simplified when specifications need not solely rely on components [14].

Interaction between components of reactive system should be captured before defining the components themselves [11]. This enables the top-down design process illustrated in Figure 1, justified as follows.

Abstract collaborative operations are used to capture initial requirements. Capturing takes place in user domain, where separate logical layers are used to formalize collaboration of each different view. Layers may also introduce real-time requirements that should be satisfied cooperatively by the involved parties. These requirements tend to represent end-to-end requirements which may not be directly satisfiable as such. Instead, complex protocols may be required in their implementation. Moreover, the requirements may also overlook some essential aspects whose incorporation in the specification has been postponed. With this approach, emphasis is placed on issues concerning what the system will do together with its environment rather than on the interfaces used to implement inter-party interactions. Furthermore, real-time requirements arising from user requirements are explicitly taken into account in a seamless fashion, as they can be included in the specification in connection with the higher-level activity.

When the necessary views have been introduced as layers, DisCo refinements support their incorporation in the same specification. In such situations, the result is the conjunction of the different views, which obviously refines all the component layers. Due to the use of property-preserving refinements, the conjunction satisfies safety properties of component layers by construction. There also is the option to use several specification branches (see e.g. [16]).

An abstraction can be omitted from an implementation provided that it can be shown that the implementation actually satisfies the abstraction. This requires proving an invariant that relates the abstraction with its implementation. In practice, this possibility to fine-tune the level of abstraction is considered particularly important for hardware-software codesign, where the levels of abstraction of different components and their interfaces may vary in different configurations. Obviously, as the level of granularity in actions decreases, there are more and more actions that have been stuttering in the top-level specification, but which have an essential role when refining the specification towards an implementation.

3.2 Real-Time Requirements

Real-time requirements need explicit design attention whenever an abstract operation attached with real-time requirements is refined towards an implementation in a fashion where stuttering actions are transformed into non-stuttering ones. In practice, this means that a longer sequence of more concrete actions is required to take place to implement a sequence given in an abstract layer.

In such situations real-time requirements given with respect to the sequence in the abstract layer may need to be partitioned into actions that are used to implement the abstraction in the conjunction. Partitioning is more crucial for bounded response requirements than minimal separation requirements; it is always more easy to consume time than to get things done in time. When requirements are partitioned, they may be strengthened at the same time. The requirements can be partitioned in a recursive fashion, resulting in the possibility to have a hierarchy of abstract requirements. This is analogous to file abstraction; the top-level abstraction is available for the user, and lower levels of abstraction are used to derive an implementation in a certain environment.

Due to the use of superposition, all real-time requirements, even those that have been implemented with lower-level ones, are always included in the specification. Therefore, when a collaborative operation has been described in detail, the real-time requirement given with respect to its abstract versions are included in the specification as an abstraction. These abstractions are essential for tracing real-time requirements between different levels of abstraction, i.e., from an abstract specification to an implementation, or vice versa.

4 Telecommand Dispatcher

As an example this section gives a formalization of a simple device that dispatches telecommands to peripherals attached to the device. The specification is based on the following user requirements:

1. Reception of telecommands is a sporadic event.
2. Telecommands can be received every 100 ms (an assumption based on the available bandwidth, for instance).
3. Available telecommands are
   (a) Turn on or off a peripheral,
   (b) Deliver a command directly to a peripheral that is turned on.
4. Access to a peripheral takes 20 – 25 ms.

The formalization is given in three steps, each of which is discussed separately. The first step provides the view the user has on the device, consisting of a simple form of telecommand handling. The second step introduces a view based on environment assumptions, and defines the commanding of the associated peripherals. The third step conjoins these two layers, resulting in a specification that satisfies the above specification.

4.1 User View

The layer presented in this subsection satisfies requirements 1 and 2 given in the above list. At this level of abstraction, telecommands are seen as record variables, and the device is introduced as a class,

$$\mathbf{type}\; TC_1 = \{\},$$

$$\begin{aligned}\mathbf{class}\; Dev_1 = \{&tc : TC;\\ &wait : boolean;\\ &d_{br1} : time\}.\end{aligned}$$

In class *Dev*, tc represents a telecommand which is currently being processed, and $wait$ is a boolean variable, initialized as true, indicating whether the device is waiting for a telecommand or not. Variable d_{br1} is used for the bounded response requirement given with respect to the execution of a telecommand. Contents of the telecommands are invisible at this level of abstraction, as they provide no significant behavioral properties. Actions are provided for receiving a new telecommand, and for reacting to it,

$$\begin{aligned} &Receive_1(tc : TC; d : Dev) : d.wait \\ &\qquad \rightarrow d.tc' = tc \\ &\qquad\quad d.wait' = false \\ &\qquad\quad d.d'_{br1} = \Delta\_on(100), \end{aligned}$$

$$\begin{aligned} &React_1(d^* : Dev) : \neg d.wait \\ &\qquad \rightarrow d.wait' = true \\ &\qquad\quad \Delta\_off(d.d_{br1}). \end{aligned}$$

4.2 Environment Assumptions

This subsection formalizes necessary environment assumptions by defining a model on how the peripheral satisfies requirement 4. Class *Peripheral* representing a peripheral contains variable $busy$, initialized as false, that indicates whether the peripheral is busy. Variables d_{br2} and d_{ms2} are used for bounded response and minimal separation requirements for the completion of the access to the peripheral, respectively. In addition, each peripheral has identity id, which is assumed to be a unique integer value. This is formalized as follows:

$$\begin{aligned} \textbf{class } Peripheral_2 = \{&busy : boolean; \\ &d_{br2}, d_{ms2} : time; \\ &id : integer\}. \end{aligned}$$

Actions are given for initiating and terminating an access to the peripheral.

$$\begin{aligned} &Initiate_2(p : Peripheral) : \neg p.busy \\ &\qquad \rightarrow p.busy' = true \\ &\qquad\quad p.d'_{br2} = \Delta\_on(25) \\ &\qquad\quad p.d'_{ms2} = \tau + 20, \end{aligned}$$

$$\begin{aligned} &Complete_2(p^* : Peripheral) : p.busy \\ &\qquad \wedge\ \tau \geq d.d_{ms2} \\ &\qquad \rightarrow p.busy' = false \\ &\qquad\quad \Delta\_off(p.d_{br2}). \end{aligned}$$

These operations now represent all accesses to the peripheral. Details of accesses could be given with additional layers.

4.3 Conjoining the Views

This subsection conjoins the layers given above in a fashion that satisfies requirements 3a and 3b. Real-time requirements given with respect to the execution of the telecommand are partitioned into the actions that represent processing and execution of the telecommand in the conjunction of the above layers.

Class *Dev* is extended with variable d_{br3} that is applied when processing telecommands,

$$\textbf{class } Dev_3 = Dev_1 + \{d_{br3} : time\}.$$

In addition, devices and peripherals are connected with relations *Control* and *Powered*,

$$\begin{aligned} &\textbf{relation } 1 \cdot Control \cdot * : Dev \times Peripheral, \\ &\textbf{relation } 1 \cdot Powered \cdot * : Dev \times Peripheral, \end{aligned}$$

which relate the device with controlled and powered peripherals, respectively.

Type TC is extended with variable *com* of an enumerated type indicating the desired operation. At this level of abstraction, we use three enumerated values, *on*, *off*, and *direct*. In addition, the telecommand has a reference to the peripheral it is intended for,

$$\begin{aligned} \textbf{type } TC_3 = TC_1 + \{&com : \{on, off, direct\}; \\ &to : integer\}. \end{aligned}$$

Of these values, *on* and *off* cause the device to switch the peripheral on and off, and $direct$ is a

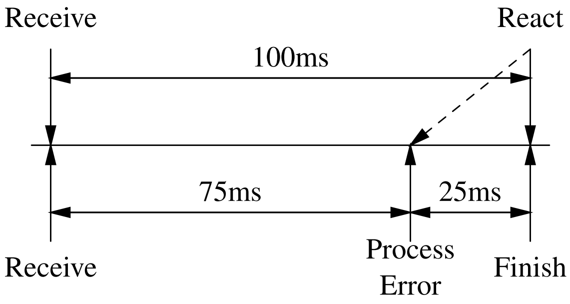

Figure 2: Partitioning of a real-time requirement

command which is dispatched directly to the peripheral, assuming that the peripheral is on.

Telecommands now take place according to the following scheme. First, action *Receive* is used to receive a telecommand. Then, the telecommand is verified to be executable, invalidating, for instance, telecommands indicating a direct command when the peripheral is turned off. For correct telecommands, an access is then made to the peripheral, formalized as action *Process*. When the access has been terminated, the telecommand is completed by executing action *Finish*. An occurrence of an error when processing the telecommand is modeled as action *Error*, in which case the peripheral is not accessed in the first place.

The situation is illustrated in Figure 2, which visualizes an execution of a telecommand in the layer introducing abstract telecommanding, and in the final layer. In the figure, only latest execution moments of actions are indicated. Of these requirements, 25 ms is an assumption made about the behavior of the peripheral, and 75 ms is a real-time requirement obtained by partitioning the time given for telecommand execution into actions used in the lower-level specification. The dashed arrow indicates the situation in case of an erroneous telecommand.

The required actions are formalized as follows. Action *Receive* is still used to receive telecommands, but it is refined to add a deadline for processing of telecommands,

$$\begin{aligned} &Receive_3(tc : TC; d : Dev) \\ &\qquad \textbf{refines } Receive_1(tc, d) \\ &\qquad \rightarrow d.d'_{br3} = \Delta\_on(75). \end{aligned}$$

Action *Process* checks that the operation indicated by the telecommand is valid, and sends the associated command to the peripheral. The action is obtained by refining action *Initiate*, which starts an access to the peripheral. The action removes the deadline indicated by variable $d.d_{br3}$, originally added in action $Receive$. This results in the following formalization:

$$\begin{aligned} &Process_3(d^* : Dev, p : Peripheral) : \\ &\qquad \textbf{refines } Initiate_2(p) \\ &\qquad \wedge\ d.tc.to = p.id \\ &\qquad \wedge\ d \cdot Control \cdot p \\ &\qquad \wedge\ (d \cdot Powered \cdot p \Rightarrow d.tc.com \neq on) \\ &\qquad \wedge\ (\neg d \cdot Powered \cdot p \Rightarrow d.tc.com = on) \\ &\qquad \rightarrow \Delta\_off(d.d_{br3}). \end{aligned}$$

Action *Finish* completes an execution of the telecommand. In this representation, the action is obtained by refining actions *React* of the telecommanding layer, and *Complete* of the peripheral layer. The resulting action also updates relation *Powered*. Deadlines added using variables d_{br3} in action *Receive* and d_{br2} in action *Initiate*, ensure that the execution moments of the action satisfy the informal specification. This results in the following formalization:

$$\begin{aligned} &Finish_3(d^* : Dev, p^* : Peripheral) : \\ &\qquad \textbf{refines } React_1(d) \\ &\qquad \wedge\ \textbf{refines } Complete_2(p) \\ &\qquad \rightarrow (d.tc.com = on \Rightarrow d \cdot Powered' \cdot p) \\ &\qquad \wedge\ (d.tc.com = off \Rightarrow \neg d \cdot Powered' \cdot p). \end{aligned}$$

In addition to actions implementing the dispatching of successful commands, there also is the possibility that a telecommand is invalid. For instance, a telecommand may indicate a peripheral that does not exist, or try to send a command to a peripheral that has been switched off. In order to recover from such situations without violating deadlines or entering a deadlock, a way to react to an erroneous telecommand is introduced,

$$
\begin{aligned}
&Error_3(d^* : Dev) : \\
&\qquad \textbf{refines } React_1(d) \\
&\quad \wedge ((\forall p : Peripheral : d.tc.to \neq p.id) \\
&\qquad \vee \\
&\qquad (\exists p : Peripheral \| d.tc.to = p.id : \\
&\qquad\qquad (\neg d \cdot Control \cdot p) \\
&\qquad\quad \vee (d \cdot Powered \cdot p \wedge d.tc.com = on) \\
&\qquad\quad \vee (\neg d \cdot Powered \cdot p \wedge d.tc.com \neq on))) \\
&\quad \rightarrow \Delta\_off(d.d_{br3}).
\end{aligned}
$$

It should be verified that the bounded response requirements given using variables d_{br2} and d_{br3} implement the one given by d_{br1}, i.e.

$$
\begin{aligned}
(\langle \mathcal{R}eceive \rangle &\leadsto \langle \mathcal{P}E \wedge \tau \leq \tau_{\mathcal{R}eceive} + 75 \rangle \wedge \\
\langle \mathcal{P}E \rangle &\leadsto \langle \mathcal{R}eact \wedge \tau \leq \tau_{\mathcal{P}E} + 25 \rangle) \Rightarrow \\
\langle \mathcal{R}eceive \rangle &\leadsto \langle \mathcal{R}eact \wedge \tau \leq \tau_{\mathcal{R}eceive} + 100 \rangle,
\end{aligned}
$$

where $\mathcal{P}E$ denotes $\mathcal{P}rocess \vee \mathcal{E}rror$. The actual proof is omitted here.

5 Conclusions

We have presented a design approach for real-time reactive systems where abstract real-time requirements and their partitioning are supported. In the beginning of the design, only properties that have explicit importance to stakeholders are included in the specification. Different views of the system can be specified separately. In later stages, conjunctions of different views derive the specification towards an implementation. When composing the conjunctions, abstract real-time requirements given in high-level views can be verified to be satisfied with concrete real-time requirements imposed on lower-level concepts. Moreover, the approach enables tracing of real-time requirements between different layers of the specification, given at different levels of abstraction.

Technically, the approach relies on logical layers as basis for specifications. With such layers, the emphasis is not on individual structural components but on their (potentially abstract) collaboration. The essential property of the approach is views that introduce behaviors of partial implementations which include real-time requirements. It is then a design decision to conjoin the layers introducing different views, and to partition abstract real-time requirements into lower-level ones.

References

[1] R. J. R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, Oct. 1988.

[2] R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. *Distributed Computing*, 3:73–78, 1989.

[3] K. M. Chandy and J. Misra. *Parallel Program Design, A Foundation*. Addison-Wesley, 1988.

[4] J. O. Coplien. Idioms and patterns as architectural literature. *IEEE Software*, 14(1):36–42, Jan./Feb. 1997.

[5] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, Aug. 1980.

[6] H.-M. Järvinen. *The design of a specification language for reactive systems*. PhD thesis, Tampere University of Technology, 1992.

[7] H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-oriented specification of reactive systems. In *Proceedings of the 12th International Conference on Software Engineering*, pages 63–71. IEEE Computer Society Press, 1990.

[8] P. Kellomäki. Verification of reactive systems using DisCo and PVS. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, number 1313 in LNCS, pages 589–604. Springer–Verlag, 1997.

[9] P. Kellomäki and T. Mikkonen. Modeling distributed state as an abstract object. In *Proceedings of the International IFIP Workshop on Distributed and Parallel Embedded Systems*, To appear.

[10] R. Kurki-Suonio and M. Katara. Logical layers in specifications with distributed objects and real time. *Computer Systems Science & Engineering*, 14(4):217–226, July 1999.

[11] R. Kurki-Suonio and T. Mikkonen. Harnessing the power of interaction. In H. Jaakkola, H. Kangassalo, and E. Kawaguchi, editors, *Information Modelling and Knowledge Bases X*, pages 1–11. IOS Press, 1999.

[12] S. S. Lam and A. U. Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, 10(4):325–342, July 1984.

[13] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[14] L. Lamport. Composition: A way to make proofs harder. Digital Systems Research Center, Tech. Note 1997-030a, Dec. 1997.

[15] T. Mikkonen. A development cycle for dependable reactive systems. In Y. Chen, editor, *Proceedings of the IFIP International Workshop on Dependable computing and its Applications*, pages 70–82, Johannesburg, 1998. University of Witwatersrand.

[16] T. Mikkonen. A layer-based formalization of an on-board instrument. Technical Report 18, Tampere University of Technology, Software Systems Laboratory, 1998. Available at `http://www.cs.tut.fi/~tjm/coala.ps`.

[17] T. Mikkonen. *Abstractions and Logical Layers in Specifications of Reactive Systems*. PhD thesis, Tampere University of Technology, 1999.

[18] T. Mikkonen. Codesign requires closed-system specifications. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, volume VI, pages 2968–2974. CSREA Press, 1999.

[19] T. Mikkonen. The two dimensions of an architecture. A position paper in the First Working IFIP Conference on Software Architecture, Feb. 1999.

[20] K. Systä. A graphical tool for specification of reactive systems. In *Proceedings of the Euromicro'91 Workshop on Real-Time Systems*, pages 12–19, Paris, France, June 1991. IEEE Computer Society Press.

[21] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.