A Development Cycle for Dependable Reactive Systems

Tommi Mikkonen
Software Systems Laboratory
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
Tel. +358 3 365 3815, Fax +358 3 365 2913
tjm@cs.tut.fi

Abstract

The development of systems consisting of a mixture of software and hardware is difficult. These systems are often developed so that hardware is defined first, and the parts implemented by software are developed after that. One drawback of this approach is that it does not focus early enough on the temporal behavior of the system as a whole. Critical issues concerning the behavior can therefore be overlooked at the beginning of the development, leading to changes late in the development, when the focus is finally set on the behavior. Similar problems can be identified in the development of complex software systems. Difficulties arising from such an approach can be overcome by developing a closed-system specification that formalizes the intended behavior early in the development. This specification can then be partitioned into components, which can be independently implemented. As the specification is closed, it also contains a description of the environment of each such component. Thus, whenever the specification is partitioned into parts that are implemented, an unambiguous specification, including the description of the environment, is available for each component. In addition, a formal behavioral specification can be used to enhance verification and validation of the resulting system.

1 Introduction

Traditionally, the development of a *reactive system*, i.e., one that is in continuous interaction with its environment, is started by fixing the underlying hardware architecture. Only after having fixed the hardware, does the interest shift to the software that is built on the hardware platform. If some system requirements are not identified early enough, radical changes may be required to the behavior of the software late in the development. In such cases, delays may be introduced to the development, and the dependability of the resulting system may be jeopardized. Similar problems can be identified in the development of complex software systems involving cooperation of several software components; the tendency is to identify processes, objects, or modules that will form the architecture. The temporal behavior of the system, however, may be ignored at the beginning of the development.

Such an approach relying on architectural modularity requires decisions with respect to the following aspects. Firstly, the required components must be unequivocally identified; their responsibilities need to be unambiguous. Secondly, inter-component interfaces must be completely defined. A commitment to these decisions results in an architecture. The behavior of the system as a whole is then defined by the behaviors of all architectural units, not vice versa. Therefore, no real definition for the temporal behavior of the system exists, until we have defined the behaviors of all independent components. Experience from similar systems increases the likelihood of succeeding in defining a satisfying behavior for each component. Lack of familiarity causes problems as the architecture is defined relying on nothing but intuition.

Alternatively, the development of reactive systems could begin by focusing on the intended behavior of the system as a whole instead of on individual components that form its architecture. Such an approach demands an unambiguous behavioral definition in the beginning of the development, and only after this is the emphasis placed on architectural definitions. As the focus is initially on the behavior, the specification should be built by using behavioral aspects as the units of modularity instead of architectural modularity based on encapsulation.

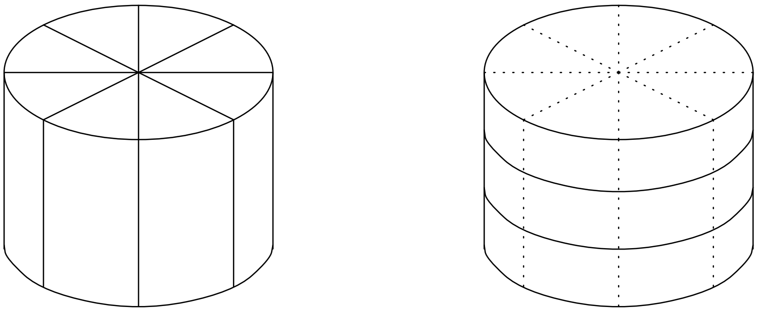

Figure 1: Architectural versus behavioral modularity

Traditionally, systems have been understood as collections of modules each of which encapsulates data and operations. This emphasizes the architecture of the system. In the alternative approach, the system can be understood as a collection of layers each of which adds new behavioral aspects to the specification. Such layers enhance the understanding of how the elements included in the specification cooperate. The two approaches are illustrated in Figure 1. Each segment in the left-hand side "cake" represents a module encapsulating a set of properties, and each layer in the right-hand side "cake" represents a behavioral aspect of the system possibly involving several implementation components. In addition, as the behavioral specification may indicate a convenient architecture, dotted lines are used for identifying potential implementation components.

In the proposed approach, we focus on the behavior of the system before an architecture has emerged. It may therefore turn out that some of the operations are not local to any of the components. Instead, one operation may affect the internal states of several components. In terms of the temporal behavior, such operations are acceptable, as they can be interpreted to represent abstractions of inter-component communication between several participants. In general, allowing operations that involve several components allows to raise the level of abstraction above programming-level abstractions that tend to be used in object-oriented specification [10]. The cooperation of components is thus highlighted, and a detailed description of how individual components implement this cooperation is omitted.

When applying behavioral modularity, it is a design decision to define the underlying hardware or software architecture. As a behavioral specification exists, the goal of the development is known. This allows dependable development of co-designs of hardware and software as well as construction of complex systems introducing several independent components implemented solely in software or hardware. In addition, distribution need not to be taken into account at the beginning of the development, as it can be regarded as a design decision that is made when the behavior of the system has been specified.

The above leads to a development cycle as defined in Figure 2. In the figure, each box labeled 'component' denotes the development of an implementation of one component. The box labeled 'implementation' denotes a complete system resulting from the development.

Assuming that a specification covering the system as a whole exists, verification and validation can also be enhanced. The behavior can be validated early in the development, and layers that structure the specification can be used to structure the validation. Formal proofs can be used to show that the specification includes certain desired properties. In addition, the specification can be used in formulating test cases to ensure that all the necessary parts are tested properly. This further increases confidence on the dependability of the resulting system.

Such a behavioral modularity may be difficult to establish using conventional methods or commonly used abstractions [10]. New approaches are therefore needed. In the following, we will introduce DisCo [7], a method that supports stepwise refinement with respect to the temporal behavior.

2 Specifying a reactive system

DisCo is a language [8] and a method [7], whose formal basis is in Temporal Logic of Actions, TLA [11]. The origins of the approach are in *joint actions* first used as abstractions of complex inter-component

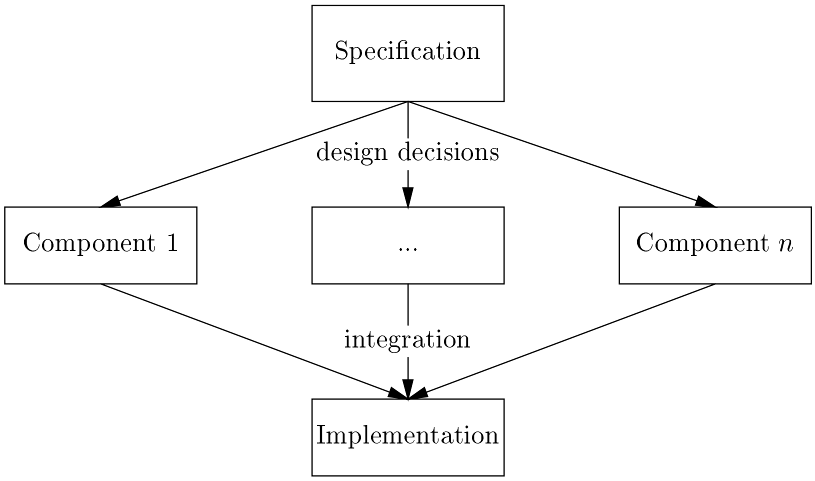

Figure 2: Enhanced development cycle

```
class Controller is
-- The name of the class is Controller.
  state *idle, used;
  -- Introduces a state machine with two states, idle is the default state.
  extend used by
  -- State used is extended by a state machine and a variable.
    state *connect, ready, disconnect;
    communicateWith: Controller := null;
    -- A reference to an instance of class Controller, default is null.
  end;
end;
```

Figure 3: A simple DisCo class

communication in [1]. Many of the ideas incorporated in the method are similar to those used in *Unity* [2]. In brief, the method defines a programming-oriented way to use the reasoning capabilities of TLA in an object-oriented framework.

The language that is used for composing specifications is textual, but graphical formalisms are used for animation in the associated tool [15]. This allows the developer to evaluate the behavior of a system when composing a specification. In addition to informal considerations and animation, computer-aided rigorous reasoning is also possible [9].

The presentation here is simplified, and many details of the DisCo language are omitted. Also, the formal background of the method is not discussed. Instead, we concentrate on how specifications are typically composed in DisCo, including the form of behavioral modularity embedded in the method.

2.1 Composing a simple specification

DisCo specifications are *systems* modeled in terms of *objects* and *actions*. Each object is an instance of a *class*, and may contain variables of basic types, references to other objects, and state machines that can be nested similarly to StateCharts [5]. As an example, a simple class, representing a connection controller of a simple router, is given in Figure 3.

Actions are inter-object operations that can be understood as multi-object methods. An action defines a set of objects that *participate* in the execution of the action in a certain *role*, an enabling *guard*, and a set of statements that may alter the local states of the participating objects. When there are several objects that can take the same role in an action, one of the possible participants is nondeterministically selected. Similarly, when there are several actions that could be executed, one of them is chosen in a nondeterministic manner. Actions that are chosen for execution are executed

```
action Start
-- The name of the action is Start.
by c: Controller is
-- One participant role c for an object in class Controller.
when c.idle do
-- Enabled when there is a Controller whose state machine is in state idle.
  -> c.used;
  -- State transition to state used.
  -> c.used.connect;
  -- State transition to state connect inside state used.
end;
```

Figure 4: A simple DisCo action

in an interleaving manner. A simple action by a single participant is given in Figure 4.

Any DisCo system introducing variables (or state machines) must also introduce all the operations that can alter the values of these variables (or states). A complete layer of properties is thus introduced in one system. Therefore, a system that contains only the class in Figure 3, and the action in Figure 4 would not introduce a way to terminate a connection. A complete DisCo system, containing actions for making a connection and closing it, is given in Figure 5. In the given system, each controller used for handling incoming traffic can connect to a controller used for outgoing traffic, and any two connected controllers can disconnect. The level of abstraction is such that all that is seen are the connections between controllers, and whether a controller is active or not. The convention is used that access to a variable that is local to a state implicitly requires the associated state machine to be in that particular state, enabling us to write the guards in a more natural form. The specification is illustrated in Figure 6, where actions are denoted as ellipses and classes as boxes.

2.2 Incrementality

It is crucial for behavioral modularity to facilitate the introduction of new behavioral aspects without compromising existing properties. In the sense of Figure 1, we need a well-defined way to add new layers without invalidating those that already exist. In DisCo, this is achieved by restricting refinements to ones obtained by *superposition* [3]. The variant of superposition adopted in DisCo [7] allows following refinements:

- New classes can be added.
- New variables can be added to existing classes.
- New actions can be added, if they do not alter values of already existing variables.
- Existing actions can be expanded, if the extension does not alter values of already existing variables.

Such refinements, whose relation to a complete system resembles *program slices* [16], guarantee that *safety properties*, i.e., properties of the form ‘nothing bad will ever happen,’ of the previous level are not compromised. However, as the enabling conditions of actions can be made more restrictive, *liveness properties*, i.e., properties of the form ‘something good will eventually happen,’ are not guaranteed to be preserved by construction. For these properties, proof obligations are raised that can be checked informally or by formal proofs.

For the example given above, a new behavioral layer can be constructed by introducing the phases of a connection that can be identified in each controller. Figure 7 gives such a refinement using the DisCo method. In it, classes representing controllers are extended by a state machine that describes their internal states in different phases of a connection. In addition, existing actions are refined and new actions introduced in order to realistically incorporate the phases into the system. In the refinement, three dots (...) are used to refer to guards and statements of the previous level.

The refinement is illustrated in Figure 8. Further refinements could be used to include other aspects in the system. For instance, the selection of a suitable outgoing controller, charging for established connections, and transmission of actual data are candidates for additional refinements.

```
system Connections is

  class InController is
    state *idle, used;
    extend used by
      whereTo: OutController := null;
    end;
  end;

  class OutController is
    state *idle, used;
    extend used by
      whereFrom: InController := null;
    end;
  end;

  action Start by ic: InController is
  when ic.idle do
    -> ic.used;
  end;

  action Connect by ic: InController; oc: OutController is
  when ic.used.whereTo = null and oc.idle do
    -> oc.used;
    ic.used.whereTo := oc;
    oc.used.whereFrom := ic;
  end;

  action Disconnect by ic: InController; oc: OutController is
  when (oc = ic.used.whereTo) or (ic = oc.used.whereFrom) do
    ic.used.whereTo := null;
    oc.used.whereFrom := null;
  end;

  action Terminate_IC by ic: InController is
  when ic.used.whereTo = null do
    -> ic.idle;
  end;

  action Terminate_OC by oc: OutController is
  when oc.used.whereFrom = null do
    -> oc.idle;
  end;
end;
```

Figure 5: A DisCo system

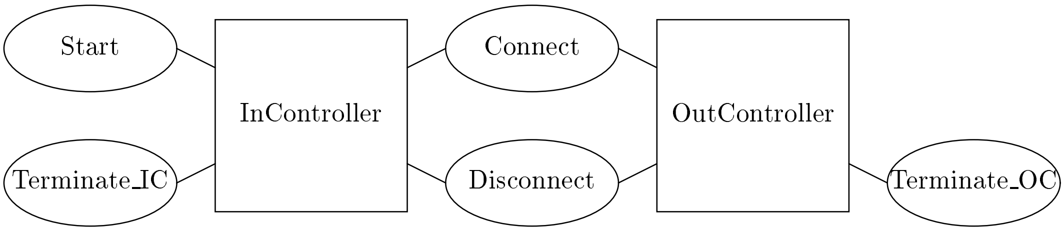

Figure 6: Illustration of a DisCo system

```
system Phases import Connections; is

  extend InController.used by
    state *connect, ready, disconnect;
  end;

  extend OutController.used by
    state *connect, ready, disconnect;
  end;

  refined Start is
  when ... do
    ...
    -> ic.used.connect;
  end;

  refined Connect is
  when ... ic.used.connect do
    ...
    -> oc.used.connect;
    -> ic.used.ready;
  end;

  action Complete by oc: OutController is
  when oc.used.connect do
    -> oc.used.ready;
  end;

  action Stop_IC by ic: InController is
  when ic.used.connect or ic.used.ready do
    -> ic.used.disconnect;
  end;

  action Stop_OC by oc: OutController is
  when oc.used.ready do
    -> oc.used.disconnect;
  end;

  refined Disconnect is
  when ... ic.used.disconnect or oc.used.disconnect do
    ...
    -> oc.used.disconnect;
    -> ic.used.disconnect;
  end;

  refined Terminate_IC is
  when ... ic.used.disconnect do
    ...
  end;

  refined Terminate_OC is
  when ... oc.used.disconnect do
    ...
  end;
end;
```

Figure 7: A DisCo refinement

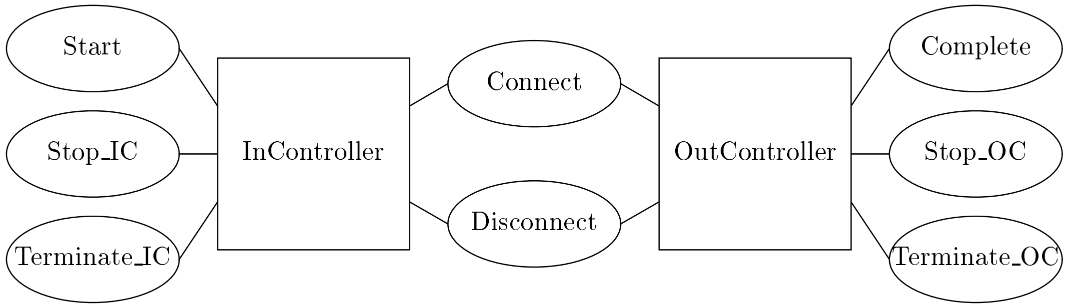

Figure 8: Illustration of a DisCo refinement

2.3 Closed versus open systems

All DisCo systems are *closed* in the sense that they always model both the system to be implemented and its environment. In contrast to closed systems, *open* systems contain only the system, and an interface that is accessible to the environment. Any open system can be modeled as a closed system by embedding it into an environment that utilizes its interface arbitrarily in a nondeterministic manner.

Although open systems by definition model only the system, assumptions concerning the environment still need to be made in order to implement an open-system specification. This can lead to a situation where different formalisms are used for defining the system and its environment assumptions. As closed systems use only one formalism, rigorous considerations with regard to specifications are more conveniently dealt with in closed systems.

The closed-system principle offers a convenient basis for behavioral modularity. Until a variable is explicitly included in the specification, it cannot affect the system. Once a variable is included in the specification, its behavior is fully defined[1] by the system, and the effect the variable has on the rest of the system can be formalized. This allows rigorous reasoning and animation upon introducing a layer, disregarding whether the specification is complete or not.

3 Design decisions

We have already discussed how to compose a specification by using behavioral modularity. Next, we make design decisions that will structure the specification into an implementable form. This includes at least the following decisions:

- Partitioning of the model to architectural components. These components define the system architecture, thus resembling traditional encapsulation.
- Definition of interfaces that the identified components use for communication. If the specification is not detailed enough with respect to interfaces, it is also a design decision to refine them.
- Selection of an implementation method. Each identified architectural component is associated with an implementation method.

In the following, each of the above decisions is discussed separately. In practice, none of them should be made independently of the others. Moreover, the design process may be iterative, taking into account cost, performance, and fabrication time of the system, obviously affected by these decisions.

3.1 Partitioning

A design decision defining implementation components of a DisCo specification defines the architecture of the system. These components can be characterized by the ability to perform operations. They may, for instance, be software processes or digital circuits. During partitioning, attention is paid on

[1] Fully defined in the sense that new assignments to the variable are impossible. Nondeterministic assignments may, however, be refined into a more precise form.

aspects like who is active in an execution, and what components are needed for an operation performed jointly by several objects.

Each component may contain objects, and be responsible for executing actions as defined in the specification. Thus, each component has a strong resemblance to architectural modules (or segments in Figure 1). However, the difference between the proposed approach and conventional techniques is that now the decisions can be based on an unambiguous behavioral specification rather than intuition and informal expectations.

Due to the closed-system principle, a DisCo specification always includes some parts of the operational environment in addition to those components that are to be implemented. During partitioning, this environment can be treated as a component or a set of components, depending on what the conceptual interpretation is. However, such components need not to be implemented, because they represent users, operators, or other physical phenomena beyond the system being developed.

In the router discussed above, the following components can be identified:

- Component *Sender* initiates a connection. It is an environment component containing no specification-level objects. The component is responsible for initiating executions of actions *Start* and *Stop_IC*. In addition, it is involved in an execution of action *Terminate_IC*.

- *Incoming* is a component handling incoming traffic. Each such component contains one instance of class *InController*. The component can initiate an execution of *Connect*, *Disconnect*, or *Terminate_IC*. In addition, the component is involved in executions of actions *Start*, *Stop_IC*, and *Disconnect* whenever the instance of class *InController* contained by the component is concerned. Notice that the component cannot execute *Disconnect* action singlehandedly, as the specification requires that connected *InController* and *OutController* are involved.

- *Outgoing* is a component handling outgoing traffic. Each such component contains one instance of class *OutController*. The component can initiate an execution of action *Complete*, *Disconnect*, or *Terminate_OC*. In addition, the component is involved in executions of *Connect*, *Disconnect*, and *Stop_OC* whenever the execution concerns the instance of class *OutController* contained by the component. The discussion given above for action *Disconnect* holds here as well.

- *Receiver* is a component that communicates with *Sender* by using the router. It is an environment component that contains no specification-level objects. The component is responsible for initiating executions of action *Stop_OC*. In addition, the component is involved in executions of actions *Complete* and *Terminate_OC*.

Figure 9 illustrates the partitioned specification. In the figure, dotted lines are used to indicate the defined partitioning.

Partitioning to components also covers aspects related to distribution. The defined approach allows that the behavior of the system is defined first, and only after that does the interest shift to the actual architecture. Therefore, it is a part of the partitioning process to identify parts that are distributed.

Each component obtained by partitioning can be further partitioned to sub-components. By applying this recursively, any level of granularity is achievable.

3.2 Defining interfaces

Partitioning identifies components of the system. For components that communicate with each other, design decisions with respect to their interfaces are essential. These interfaces may be fixed in advance, but in many cases there are several alternatives.

For each component, the selection of an interface must be made in a manner that allows the satisfaction of the specification. This may require a lot of work, as what appears to be one action in the specification may be an abstraction of the cooperation of several components in the implementation. In such a case, interfaces of the associated components need to be refined to include negotiation whether to execute an action or not, as discussed in [14]. This also applies to the interfaces that can be accessed by the environment, as in practice, an interrupt handler or an event-handling mechanism may be needed for accessing the system even if they do not exist in the specification.

For the case discussed in this paper, the interface used for inter-component communication could, for instance, consist of asynchronous messages. However, actions involving several components cannot

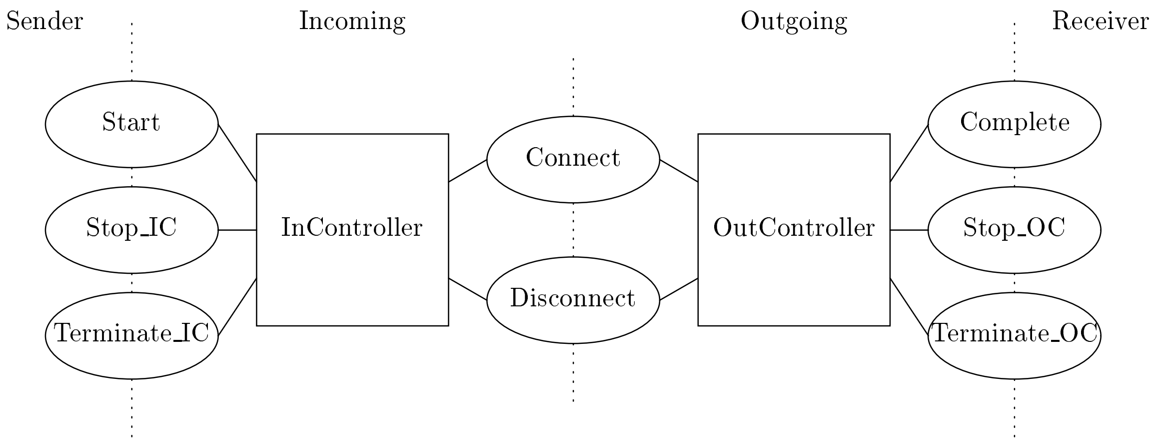

Figure 9: A partitioned DisCo specification

in general be directly implemented in terms of asynchronous messages. Therefore, the interfaces associated with the components should be refined. This task, however, falls outside the scope of this paper, and is discussed e.g. in [14].

3.3 Selection of implementation method

A design decision concerning an implementation method is needed for each identified component. Obviously, if some part of the system is implemented as a dedicated circuit, that particular part is not likely to change in the production phase. On the other hand, if some part of the system is identified as change-prone when composing the specification, it is probably a good idea to implement that part by software paying special attention to maintainability. Such decisions duly effect the development, if several releases are planned, each of which introduces a set of new features.

Conventional methods like C++ [4], for instance, are possible implementation methods. For the example in this paper, the following implementation techniques could be selected.

- Component *Sender* represents the environment. Its implementation is therefore unnecessary.
- Each *Incoming* is a part of the system. This component could be implemented by hardware or by software, depending on other requirements associated with the component. Here, C++ is selected for the implementation method.
- Each *Outgoing* is a part of the system. Again, C++ is selected for the implementation method.
- Component *Receiver* represents the environment, and therefore, it need not to be implemented.

4 Implementing components

After committing to the design decisions discussed above, the task of composing an actual implementation can be considered as a set of independent tasks. As a closed-system specification is available, an unambiguous behavioral specification exists for all the components. Furthermore, as interfaces are refined to an implementable level, each component has a completely defined interface. Different components can therefore be implemented independently. As the contents of the components correspond to collections of DisCo objects and actions, their implementation is relatively straightforward, although several implementation schemes exist [12].

It is expected that the specification is implemented by hand, although there have been some experiments for automatic code generation directly from a DisCo specification [13]. One drawback of such an approach is that the level of detail needed for realistic code generation makes the specification very complex, with possibly the most essential aspects of the specification being lost among implementation details.

Implementation of a component may be a project of its own. In this paper, the point is not the development of an implementation, but the possibility of developing a behavioral specification of the system, and only after that concentrate on issues that concern individual components. Topics associated with the development of an actual implementation therefore fall outside the scope of this paper.

5 Verification and Validation

So far, we have described the development of systems based on cooperating components. However, the dependability of a reactive system is not guaranteed by a well-defined development approach alone. A way to verify and validate the resulting system is also needed. The following subsections discuss implications of the approach, including early validation of the specification, verification based on formal aspects of the specification, and conventional testing.

5.1 Early validation

During the early phases of development, a precise executable specification is composed. This allows early validation of the specification by animation by using computer-aided tools [15]. This facility is considered important for practical usage, as implied in [6].

The use of behavioral modularity gives a structure to the specification. Each layer can be considered as a projection of the complete system, similarly to program slices [16]. Validation can therefore be carried out by focusing on one layer at a time, enabling considerations that still concern the temporal behavior of all components of the system, but concentrate on a subset of features. Furthermore, layers that are independent from one another can be validated separately. In terms of the example, it is possible to validate that connections are formed correctly, and only after that shift the focus on considering whether the phases of a connection behave in a satisfying manner.

The mechanism used for expressing inter-process communication can be understood as an abstraction of commonly available implementation techniques. Thus, the modeling is liberated from programming-level abstractions, as analyzed in [10]. The specification can therefore set the emphasis on the essential validation aspects, and overlook design issues.

5.2 Formal verification

As a formal behavioral specification is composed, it is possible to use a rigorous approach for verifying that a design satisfies the specification, or that the specification does not compromise some crucial properties. Furthermore, the use of behavioral modularity allows verification of safety properties layer by layer. Since the refinement steps always preserve safety properties, it is obvious that a proven safety property is not violated in later refinement steps. For cases where interfaces are refined, additional proofs may be necessary [14].

Proving that an implementation as a whole is correct with respect to a specification is also possible. In practice, this may be too laborious, as a detailed formal model of the implementation is necessary. Even when provided with such a model, verification and validation only address the model, not the real system. A proof can therefore only show that a model of an implementation satisfies the specification, not that the actual implementation is correct.

Details of proof techniques fall beyond the scope of this paper. Formal verification of DisCo specifications is discussed in more detail in [9].

5.3 Testing

In accordance with conventional approaches, testing of the resulting system is also necessary. Two kinds of testing can be identified. *Component testing* aims at verifying that design decisions are correctly taken into account in the implementation phase of each component. *System testing* verifies that all the components cooperate as defined in the specification. A V-model associated with the approach, including testing as defined in this subsection, is illustrated in Figure 10. As described

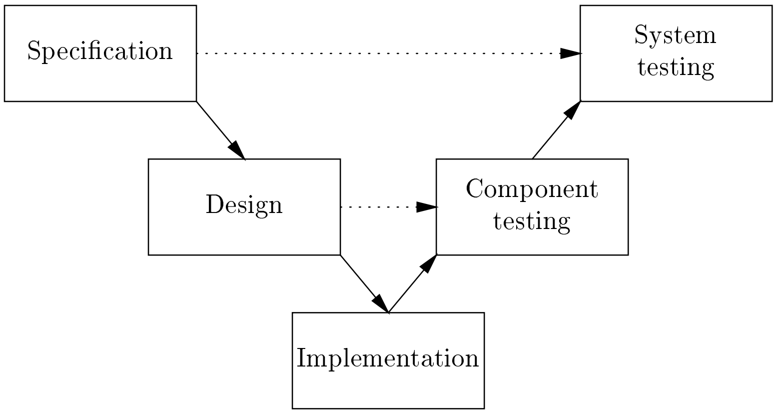

Figure 10: A V-model

above, the box labeled 'implementation' may be a large project, and therefore include some testing of its own.

Component testing verifies that the decisions made in the design phase have been carried out as planned, and that functionalities allocated to components satisfy the specification. This also applies to verification of the refined interfaces. For the example discussed in this paper, test cases are needed that verity the functionalities and interfaces associated with components *Incoming* and *Outgoing*.

System tests aim at ensuring that the implementation as a whole is composed according to the specification. Because the specification contains a precise behavioral description of the system, it should be able to simulate the behavior of the real system. Thus, any test case can be rerun by using the specification, and the results should be the same.

The fact that there is a behavioral specification allows us to measure the coverage of system testing. It is therefore possible to state what percentage of the specification was covered by the test cases, and whether or not there are situations that have not been completely covered[2]. This obviously increases confidence on the dependability of the system.

The use of an automatic code generator could lead to a situation in which testing could be totally omitted. This, however, is not considered a realistic option at the current phase of this work. A potential use for such a code generator is to use it to generate a simulator for the parts that have not yet been implemented. Each component could then be separately tested as a prototype that contains the component implemented by hand, and some automatically generated code. This would enable early verification and validation of implemented components in a real[3] environment.

6 Conclusions

In this paper, an alternative development cycle for dependable reactive systems has been introduced. The cycle enhances understanding of the temporal behavior of the system early in the development, increasing the confidence that the resulting system will be satisfactory. That said, well-designed architectural encapsulation can be understood as a special case of behavioral modularity, as it gathers operations and data into smaller modules, each of which is easier to understand and implement than the system as a whole. A closed-system view of such a situation is illustrated in Figure 11, which also illustrates the inter-component interface between the two combined components.

The approach is not tied to any particular fabrication technique. Instead, it makes no difference which parts will be implemented in hardware, and which in software. This leads to a natural approach to co-designs, and also allows all-software and all-hardware development. Moreover, as it is a design step to define the interfaces, it is possible to fine-tune the system in the design phase.

[2]Obvious restrictions exist, if the specification allows an infinite number of different executions.

[3]Real in the sense that the same specification is applicable to both automatically generated code used in the prototype and the actual implementation.

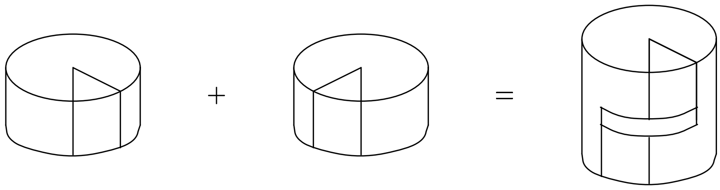

Figure 11: Components as units of behavioral modularity

The use of an unambiguous behavioral specification also enhances verification and validation. The defined approach allows early validation by animations as well as rigorous reasoning when composing the specification. In addition, the specification can be used to support testing of an implemented system.

Backed by the given discussion, and by encouraging industrial feedback obtained from a reverse-engineering approach to a real multi-processor system [6], we argue that the defined approach increases the confidence on the development as well as the resulting system. This increase should be sought, in particular, when developing reactive systems whose dependability is critical.

References

[1] R. J. R. Back, R. Kurki-Suonio, Decentralization of process nets with a centralized control. *Distributed Computing* 1989, 3: 73–87. An earlier version in *Proc. 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, 1983, 131–142.

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

[3] E.W. Dijkstra, C.S. Scholten, Termination detection for diffusing computations. *Information Processing Letters 11*, 1 (August 1980), 1-4.

[4] M.A. Ellis, B. Stroustrup, *The Annotated C++ Reference Manual.* Addison Wesley, 1991.

[5] D. Harel, STATECHARTS: A visual formalism for complex systems. *Science of Computer Programming 8*, 1987, 231-274.

[6] S. Isojärvi, DisCo and NOKIA: Experiences of DisCo with modeling real-time system in multiprocessor environment. *FMEIndSem'97*, Otaniemi, Finland, February 20, 1997.

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

[8] H.-M. Järvinen, *The Design of a Specification Language for Reactive Systems.* Doctoral dissertation, Tampere University of Technology, 1992.

[9] P. Kellomäki, Verification of reactive systems using DisCo and PVS. *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods* (Eds. J. Fitzgerald, C. B. Jones, P. Lucas), Springer-Verlag LNCS 1313, 1997, 589-604.

[10] R. Kurki-Suonio, T. Mikkonen, Liberating object-oriented modeling from programming-level abstractions. *Workshop on Precise Semantics for Object-Oriented Models (ECOOP'97)*, to be published by Springer-Verlag.

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

[12] T. Mikkonen, On implementing object-oriented specifications given as closed systems. In *Proc. Conceptual Modeling and Object-Oriented Programming*, Finnish Artificial Intelligence Society, November 5, 1993.

[13] T. Mikkonen, An experimental code generator for implementing formal specifications given as closed systems. In *Proc. of the Fourth Symposium on Programming Languages and Software Tools*, Visegrad, Hungary, June 9-10, 1995.

[14] T. Mikkonen, *Implementation of Reactive Systems Based on Closed-System Specifications.* Licentiate thesis, Tampere University of Technology, 1995.

[15] K. Systä, A graphical tool for specification of reactive systems. In *Proc. Euromicro'91 Workshop on Real-time Systems*, Paris, June 12-14, 1991.

[16] M. Weiser, Program slicing. *IEEE Transactions on Software Engineering*, Vol. SE-10, No 4, pp. 352-357, 1984.