DERIVING MESSAGE PASSING PROTOCOLS FROM COLLECTIVE BEHAVIOR

Pertti Kellomäki

1. Introduction

Many embedded systems applications involve distributed co-operating components. For cost reasons, they must be designed to use computational and communication resources as sparingly as possible. To achieve this, carefully hand-optimized solutions are often used. However, reasoning about such solutions at the implementation level using deductive techniques is like trying to verify the object code produced by an optimizing compiler: much of the structure and abstractions present at the high level of design have been lost. Recovering abstractions can be quite expensive, necessitating the formulation and verification of large strengthening invariants.

In this paper we hope to demonstrate that verifiability and efficiency are not mutually exclusive. One can write specifications that convey the designer's intent at an abstract level and are amenable to formal verification, and at the same time yield optimized implementation level solutions. Implementation level efficiency does not need to suffer from the abstractions built into a specification.

We argue that both design and verification are easier with a methodology that starts with the collective behavior and explicitly represents the underlying abstractions. When concurrent activities involve multiple interacting objects, coordination of the activities is difficult. In the telecommunications field, this *feature interaction* problem is a well-known source of difficulties. We try to avoid these problems by starting with a specification of the high level collective behavior that makes the atomicity of high level operations explicit. The paper gives methodological guidelines to help a designer in deriving low level message passing protocols from high level specifications of collective behavior. We also show how the methodology is used in a derivation and verification of a protocol achieving atomic insertions and deletions in a distributed linked ring, borrowed from [PD96].

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: [10.1007/978-0-387-35409-5_23](https://doi.org/10.1007/978-0-387-35409-5_23)

The rest of the paper is structured as follows. In Section 2 we describe the use of superposition and the DisCo specification method. Section 3 describes the specification methodology for deriving a message passing implementation from a specification of collective behavior. An example derivation is given in Section 4, related work is reviewed in Section 5, and conclusions are drawn in Section 6.

2. The DisCo approach

In this section we describe the joint action [BKS88,BKS89] approach to specification and the use of superposition, which form the foundations of the DisCo [WWW00,JKSSS90,JKS91,KS96] specification method.

2.1. Joint actions

A *joint action* is a formula for a multi-party operation that describes a step in a computation, involving a synchronization of the *participants* of the action. An action may have *parameters*, which are similar to participants except that they denote values, not objects with mutable state. The values assigned to the variables of the participants in an execution of an action may freely refer to the attributes of the other participants and the parameters of the action.

A joint action defines inter-object cooperation at a high level of abstraction, where the focus is lifted from communication details and the behavior of individual objects to the collective behavior. The states of the objects that do not participate in the execution of an action remain unchanged. Parallel activities are modeled with an interleaving semantics.

The following specification illustrates how classes and actions are defined using the DisCo syntax:

```

system L1 is
  class C is x : integer; end;
  action inc(n : integer) by a, b : C is
    when a.x < b.x do a.x := b.x + n; b.x := a.x + n; end;
end;

```

If a combination of objects satisfying the guard of the action can be found, the action is *enabled* for the objects and can be executed. The choice of which enabled action to execute and for which combination of participants and parameters is nondeterministic.

2.2. Superposition

Stepwise refinement in the DisCo approach is based on *superposition*, where new state variables are incrementally added to a specification until the desired level of detail is reached. The actions of the specification being refined may be augmented with assignments to the new state variables, but new assignments to state variables

introduced earlier may not be given. The initial condition and the guards of the actions may be strengthened.

Superposition preserves safety properties by construction, since all assignments to a state variable are given when the state variable is introduced. DisCo specifications are thus *closed world* specifications.

Nondeterministic action parameters can be used for modeling values determined by state variables not visible at a given level of abstraction. By augmenting actions with new constraints on the values the parameters may assume, one can constrain the nondeterminism at a later stage.

The following layer illustrates how superposition is done in the DisCo language.

```

system L2 import L1; is
  class stepper is step : integer; end;
  extend C by counter : integer; end;

  refined inc by ... s : stepper is when ... n = s.step do
    ... s.step := s.step + 1;
    a.counter := a.counter + 1; b.counter := b.counter + 1;
  end;
end;

```

The ellipsis “...” denotes parts from the action being refined. Superimposing *L2* on *L1* is expressed in DisCo by *importing* specification *L1* into *l2*.

3. Specification methodology

In this section we describe how an abstract specification of the collective behavior of a distributed system can be systematically transformed into an implementable specification that employs message passing.

The most abstract specification gives the effects of the operations on the objects of the system using joint actions. At this level of abstraction, an action can access the attributes of all the participants, irrespective of whether it is physically possible to access the objects simultaneously in the actual system.

In the next step, distribution is introduced as abstractly as possible. For each class *C*, whose attribute *X* needs to be accessed remotely, we introduce a class *mC* of mobile objects. Classes *C* and *mC* are augmented with new variables *x* and *valid*. The value of *X* is stored in either *C.x* or *mC.x*, as indicated by *valid*. To associate an object of class *mC* with an object of class *C*, class *mC* contains a reference *represents*. The invariant that allows treating *X* as a “shadow” variable that need not be explicitly represented in an implementation is

$$\begin{aligned}
 &\forall c : C, mc : mC :: mc.represents = c \Rightarrow \\
 &\quad c.valid \Rightarrow c.X = c.x \\
 &\quad \wedge mc.valid \Rightarrow c.X = mc.x.
 \end{aligned}$$

The mobile object can be thought of as a representative of the corresponding immobile object. Accessing a shadow variable of an object does not require synchronization with the object, provided that its representative is present.

What distinguishes our approach is that at the specification level, the mobile objects have mutable state. Instead of sending and receiving immutable messages, an object communicates with a remote object by assigning values to the variables of the representative of the remote object. Modeling distribution in this fashion introduces the essential aspects of distribution (asynchronous changes to state variables) while keeping the model as simple as possible. The simple model means that deductive formal verification of the invariant is almost trivial.

Introducing representative objects differs from ordinary data abstraction in that it is the location, not the representation of data that is being abstracted. Once the representative objects have been introduced, we can use data abstraction to derive the implementation level messages. The step from the abstract behavior to representative objects is standard enough that it could be supported by parametric superposition steps [KM00]. The linking invariant could then be verified once for the parametric layer, and reused by verifying the assumptions of the layer.

The resulting specification describes a very nondeterministic system. The mobile objects are free to roam about and interact with other objects at will. In order to constrain this nondeterminism, we introduce a state machine in the representative objects. When the *valid* attribute of an object is false, the state machine is in a state labeled *no_message*. For each interaction we wish the object to participate in, we introduce a corresponding state in the state machine, and refine the corresponding action to require that the representative object is in a suitable state.

The attributes *mc.valid*, *mc.represents*, and *mc.x* can be made shadow variables by using the state machine. The state machine itself implements *mc.valid*, if we arrange the state machine to exit state *no_message* in every action where *valid* is assigned the value true, and enter the state *no_message* in every action where *valid* is assigned the value false. For *mc.represents* and *mc.x* to be implemented, each of the states of the state machine needs to have attributes from which they can be computed. Since *mc.x* is not accessed when *mc.valid* is false, *mc.x* can be set to some known constant value when *mc.valid* is set to false, so only *mc.represents* needs to be implemented in state *no_message*.

For example, if we wanted to implement a client-server protocol where clients send requests to servers, we would introduce a state machine with states *no_message*, *request*, and *reply*. State *request* would contain attributes *from* (implementing *mc.represents*) and *current_x* (implementing *mc.x*), and state *reply* would contain attributes *to* (implementing *mc.represents*) and *new_x* (implementing *mc.x*). State *no_message* would contain attribute *from*. The following table illustrates how *C.X* is implemented by the lower level variables:

C.X	implemented by C.x	
	implemented by mC.x	implemented by mC.request.current_x
		implemented by mC.reply.new_x

Figure 1. Overview of the specification

The final step to a message passing implementation is to observe that we can interpret state *no_message* as the nonexistence of a message and the rest of the states of the state machines as messages. A state transition of the state machine corresponds to receiving a message and sending a new message. Since the attributes contained in the states are only assigned to when the state is entered, the messages do not need to be mutable.

The construction outlined above results in a correct but potentially inefficient implementation. It leaves some aspects underspecified, e.g. how messages find their way to their recipients. These aspects can be specified using further superposition steps, with the guarantee that the steps cannot invalidate the correctness properties provided by the higher levels of specification. The derived protocol can also be optimized by means of verifying additional invariants, as illustrated in the next section.

4. An example: distributed Ring

The example presented in this section is based on the description in [PD96]. The objective of the protocol is to maintain a singly linked distributed ring of cells (processors interested in a particular cache line in a multiprocessor system). A cell not in the ring may request admission to the ring, and a cell in the ring may request to be removed.

Figure 1 depicts the structure of the specification. The three layers *abstract*, *distribution* and *messages* correspond to the layers described in Section 3. Layer *add_details* specifies further details of adding a cell. Layer *token* describes how activities are coordinated using a token, and layer *delete_details* specifies how this coordination is applied to deletions. Layer *unoptimized* merges the two branches into a specification of a complete but inefficient protocol. Layer *optimized* specifies a more efficient protocol where a message and some fields of messages have been shown redundant by verifying an invariant.

We will outline the specification here, the full specification can be found at URL <http://disco.cs.tut.fi>.

4.1. The collective behavior

The specification *abstract* of the collective behavior gives *add* and *delete* as atomic operations on the ring in terms of variable *anext*, which is the specification level variable linking cells in the ring. The specification is short enough to be given in its entirety:

```

system abstract is
  class cell is anext : cell; end;

  action add by prev, c : cell is when c.anext = c
  do c.anext := prev.anext; prev.anext := c; end;

  action delete by prev, c : cell is when prev.anext = c
  do prev.anext := c.anext; c.anext := c; end;
end;

```

The specification does not constrain where cells may be added or removed, nor does it specify when these actions should take place. This is intentional: the specification only gives the essential core behavior.

4.2. Representatives

Layer *distribution* introduces the representative objects. The layer makes *anext* a shadow variable, implementing it by the state variable *next*, as explained in Section 3. The *cell* class is augmented with new state variables, and a new class *representative* is introduced:

```

extend cell by valid : boolean; next : cell; end;
class representative is represents : cell; valid : boolean; next :
cell; end;

```

New actions *send_representative* and *receive_representative* are introduced, and actions *add* and *delete* are augmented to require a representative of the requesting object as a new participant, admitting a distributed implementation. The refinement of *add* illustrates how this is accomplished in DisCo:

```

refined add by ... r : representative is
when ... prev.valid and r.valid and r.represents = c
do prev.next := r.next; r.next := prev.anext; ... end;

```

4.3. Implementation level messages

Layer *messages* introduces a state machine in the representative objects as follows:

```

extend representative by

```

```

state *no_message(from_c : cell), add_request(from_c : cell;
current_next : cell), add_reply(to_c : cell; new_next : cell),
    delete_request(from_c : cell; current_next : cell),
    delete_reply(to_c : cell; new_next : cell);
end;

```

Two refinements of action *send_representative* are introduced: *send_add_request* that sends a representative in state *add_request*, and *send_delete_request* that sends a representative in state *delete_request*. Action *receive_representative* is refined similarly. The variables contained in the state items implement variables *valid*, *represents* and *next*.

4.4. Head cell

We next refine messages relating to addition and deletion in separate branches of specification. In layer *delete_details*, a dedicated head cell is introduced, and class *cell* is augmented with an attribute *my_head*. State *add_request* is augmented with *to_c*, the address of the recipient of the address, and action *send_add_request* is refined to send requests to the cell whose address is found in *my_head*.

4.5. Token

Layer *token* introduces class *token* and actions to start and end an activity that can be used for mutual exclusion. The safety property provided by the layer is that actions *start_activity* and *stop_activity* occur in a strictly alternating sequence:

```

system token is
  class token is state *no_activity, activity_in_progress; end;
  assert single_token is  $\forall t1, t2 : \text{token} :: t1 = t2$ ;
  action start_activity by t : token is when t.no_activity
  do  $\rightarrow t.\text{activity\_in\_progress}$ ; end;
  action stop_activity by t : token is when t.activity_in_progress
  do  $\rightarrow t.\text{no\_activity}$ ; end;
end;

```

4.6. Coordinating deletions

Deletions from the ring must be coordinated, because deletion requires cooperation from the predecessor in the ring. In order for deletion to succeed, the predecessor must be in the ring. Layer *delete_details* utilizes the token to this end. Composing *send_delete_request* and *start_activity* using the *combined* clause composes the actions from the imported layers into one atomic action. The safety property from layer *token* ensures that actions *c_send_delete_request* and *c_delete* alternate:

```

system delete_details import messages, token;
combined c_send_delete_request
  of messages.send_delete_request, token.start_activity;

```

combined *c_delete* of *messages.delete*, *token.stop_activity*; **is**
details omitted end;

4.7. The unoptimized protocol

The separate branches of specification are composed in layer *unoptimized*, which simply imports the detailed specifications of add messages and delete messages. The only new element is the initial condition stating that initially the head cell is in possession of the token.

system *unoptimized import* *add_details*, *delete_details*; **is**
initially *head_has_token* **is**
 $\forall hc : \text{cell}; t : \text{token} :: hc.my_head = hc \Rightarrow t.no_activity.to_c = hc;$
end;

4.8. The optimized protocol

The final layer specifies the optimized protocol by means of additional invariants. The invariants justify the omission of certain objects and attributes from an implementation.

The first invariant concerns the token. The states *no_activity* and *activity_in_progress* were used in layer *token* to specify the mutual exclusion property. However, when it is composed with the specification of delete messages, the following invariant holds in the composed system:

$$\forall r : \text{representative}; t : \text{token} :: \\ r.delete_request \Rightarrow t.activity_in_progress.$$

The invariant allows us to implement the token only when it is in state *no_activity*.

The implicit token is not present in the implementation level description of the protocol. However, having it as an explicit object in the specification makes it possible to address the concern of mutual exclusion in an independent layer, and to combine it with a specification of messages relating to deletion.

The next invariant allows us to omit the *new_next* field of a delete reply. We observe that the value in *delete_reply.new_next* will always be equal to the recipient of the reply, expressed as

$$\forall r : \text{representative} :: r.delete_reply \Rightarrow \\ r.delete_reply.new_next = r.delete_reply.to_c.$$

This invariant justifies omission of the *new_next* field from a delete reply message.

Not surprisingly, the protocol derived here is essentially the same as described in [PD96]. However, we believe that the derivation from collective behavior makes the specification easier to understand. It certainly makes it easier to verify than the original implementation-level description.

5. Discussion

Since real systems are mostly too complicated to be verified at the implementation level, abstraction has been one of the central themes in verification (e.g. [BH99,CGL94,LGSBB95,LS84]). The standard way of using abstraction is to provide an *abstraction function* (*abstraction relation*) that links states in an abstract specification with states in a more concrete specification. The abstraction function is chosen in such a way that verifying a property for the abstract specification implies that it also holds for the more concrete specification.

Our approach differs from the usual use of abstraction in that we do not use an explicit abstraction function. When superposition is used, state variables of the abstract specification are also present in the more concrete specification, so instead of an abstraction function linking variables in two separate specification, we link the abstract and concrete variables with an invariant in one specification.

Verification based on explicit or implicit state space exploration has made impressive progress in the recent years. However, it appears that there are still difficulties with verification of systems where the number of objects is arbitrary and the identities of the objects cannot be abstracted away [PD97]. Deductive techniques such as mechanical theorem proving do not share this difficulty.

Often one of the main problems in verifying a system is to invent the correct abstractions. We try to alleviate this problem by outlining a systematic methodology for building useful abstractions for a specific class of systems. Expressing the abstractions as first class entities in the concrete specification helps in understanding the low level behavior in terms of the higher level abstractions. Some properties of the system are also nicely captured by the use of object containment. Park and Dill [PD96] verify the linked ring protocol described in this paper and comment that they have chosen to represent the network “in a non-obvious way” so that there is one variable per cell to hold a message pertaining to the cell. They then prove that the network can indeed be represented in this way without messages being overwritten. In our formulation, this property is a direct consequence of messages being represented as states of a state machine contained in a representative object.

6. Conclusions

We have demonstrated how message passing protocols can be systematically derived from descriptions of the high level collective behavior. Such a derivation results in an underspecified description of the protocol that can be further refined using superposition steps that preserve safety properties by construction. The derivation thus provides the core correctness properties of the protocol, with the guarantee that the mechanisms introduced in later steps cannot invalidate them. The protocol can be optimized by means of verifying additional invariants.

Verification of the derived protocols is considerably easier than verifying equivalent implementation-level specifications. This is due to the abstractions built into the specification by the superposition process. Some parts of the derivation can be supported by parametric superposition steps.

We believe that our approach provides considerable advantages when the system being specified involves concurrent interactions between multiple objects. In this domain, specifying the collective behavior first helps to avoid accidental interference of the individual interactions.

Acknowledgments

This research was supported by the Academy of Finland, grant 757473. Some of the work was carried out while the author was visiting the university of Dortmund on a grant from the Academy of Finland.

References

- [BH99] Ramesh Bharadwaj and Constance L. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Software Engineering: An International Journal*, 6(1):37-68, January 1999.
- [BKS88] R. J. R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513-554, October 1988.
- [BKS89] R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with a centralized control. *Distributed Computing*, (3):73-87, 1989.
- [CGL94] E. Clarke, D. Grumberg, and D. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512-1542, 1994.
- [JKS91] Hannu-Matti Järvinen and Reino Kurki-Suonio. DisCo specification language: marriage of actions and objects. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 142-151. IEEE Computer Society Press, 1991.
- [JKSS90] 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.
- [KM00] Pertti Kellomäki and Tommi Mikkonen. Design templates for collective behavior. In E. Bertino, editor, *Proceedings of ECOOP 2000*, volume 1850 of *Lecture Notes in Computer Science*, pages 277-295. Springer, 2000.
- [KS96] Reino Kurki-Suonio. Fundamentals of object-oriented specification and modeling of collective behaviors. In H. Kilov and W. Harvey, editors, *Object-Oriented Behavioral Specifications*, pages 101-120. Kluwer Academic Publishers, 1996.
- [LGSBB95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design: An International Journal*, 6(1):11-44, January 1995.
- [LS84] Simon S. Lam and A. Udaya Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, 10(4):325-342, July 1984.
- [PD96] S. Park and D. L. Dill. Protocol verification by aggregation of distributed transactions. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 300-310. Springer Verlag.
- [PD97] Fong Pong and Michel Dubois. Verification techniques for cache coherence protocols. *ACM Computing Surveys*, 29(1):82-126, March 1997.
- [WWW00] The DisCo project WWW page. At URL <http://disco.cs.tut.fi/> on the World Wide Web, 2000.