

# Formal Aspects for Distributed Systems

Pertti Kellomäki, `pk@cs.tut.fi`  
Institute of Software Systems  
Tampere University of Technology  
Finland

January 28, 2002

## Abstract

We argue that superposition and the joint action style of specification are well suited for the aspect-oriented formal specification of distributed systems. Superposition steps structure a specification according to behavior instead of implementation level components. Superposition also makes it possible to verify temporal safety properties and refine and compose specifications in a way that preserves these properties.

## 1 Introduction

Much of the complexity of distributed systems stems from the interplay of different mechanisms for atomicity, fault tolerance, etc. Because of performance considerations many distributed algorithms contain optimizations to reduce the amount of data transferred. Unfortunately these optimizations can make such algorithms hard to understand and verify. Ideally one should be able to specify the mechanisms relatively independently, and to compose them to produce specifications of efficient implementations.

Correctness of distributed systems is often expressed in terms of their temporal properties<sup>1</sup>. A specification method for distributed systems should thus allow for convenient reasoning about temporal properties, and help in modularizing both specification and verification in a way that aligns with the temporal properties. We identify the following needs.

1. Distributed mechanisms require the cooperation of several objects, so a single syntactic unit of modularity should be able to introduce data and operations into multiple implementation level components.
2. It should be possible to specify independent aspects of collective behavior in parallel branches of specification, and merge the branches to form a composed specification.
3. Because formal verification of temporal behavior is expensive, temporal properties should be preserved both in refinement and composition.
4. For the same reason it is desirable to be able express and verify aspects in an abstract form in order to reuse verification effort. It should thus be possible to design and verify aspects of collective behavior independently of their deployment.

Needs 1–3 have been addressed in the long term research project DisCo [1] and the associated specification language [6, 5]. The approach is based on stepwise refinement using superposition and joint actions (abstractions of distributed cooperation). To address the last item in the wish list, we have designed and implemented a formal specification language Ocsid, an experimental

---

<sup>1</sup>Temporal properties in the temporal logic sense, i.e. ordering of events. We do not consider real-time properties here, even though they can also be specified using superposition (see e.g. [10]).

```

specification list is
  class node is
    NEXT : ref node;
  end;
  action delete by toDelete, pred : node is
  when pred.NEXT = ref(toDelete) do
    pred.NEXT := toDelete.NEXT;
    toDelete.NEXT := ref(toDelete);
  end;
end;

```

Figure 1: A high level specification of deletion from a distributed list

variant of the DisCo language. The language allows us to specify and verify aspects of collective distributed behavior and to compose aspects into specifications in such a way that the aspects are woven together. The language has a formal semantics given in the typed higher order logic of the PVS [11] theorem prover, which allows for fully mechanized reasoning about Ocsid specifications [9].

## 2 Joint Actions and Superposition

In the *joint action* [2, 3] style of specification, distributed systems are specified using atomic guarded parallel assignments involving multiple objects. The formal basis is linear time temporal logic, and behavior is specified in terms of *state variables* that reside in objects. The structure of objects is dictated by *classes*.

A system described by a joint action specification starts in some initial state satisfying the initial condition of the specification, and proceeds to execute enabled actions one at a time. If more than one action is enabled (i.e. a combination of objects for which the guard evaluates to true exists), one of the actions is nondeterministically selected.

Figure 1 shows an example of a joint action specification, a simplified version of an example borrowed from [12]. It describes how objects of class *node* are arranged in a singly linked ring, and how a node is deleted from the ring. Identifiers *toDelete* and *pred* in action *delete* are formal *roles* in which objects may *participate*.

The initial configuration of the nodes would be specified by an initial condition which we omit in the interest of brevity. In order to further reduce the number of irrelevant details, we only consider deletion. Insertion of new nodes would be specified similarly.

Joint actions in a high level specification may imply synchronizations that are not directly possible in an actual implementation, but are useful at the specification level to express distributed cooperation. Stepwise refinement is used to introduce lower level mechanisms that implement the synchronizations. For example, action *delete* in Figure 1 cannot be directly implemented in a distributed system, because it needs to assign to the *NEXT* attribute of two objects. Our strategy is to introduce new variables from which *NEXT* can be calculated, and consequently omit *NEXT* and the synchronizations needed for accessing it from the implementation.

*Superposition* is a simple form of stepwise refinement, where the only transformation is to add new structure to a specification. An Ocsid superposition step may introduce new state variables into classes, strengthen the guards of actions, and introduce assignments to the newly introduced state variables. Previously introduced variables cannot be assigned to, which preserves safety properties<sup>2</sup> by construction.

---

<sup>2</sup>Safety properties in the temporal logic sense: something bad never happens. This is in contrast to liveness properties: something good eventually happens.

In our example, we superimpose two implementation level mechanisms on the high level specification: an implementation of atomic assignment using a state machine and the exchange of request and reply messages, and coordination implemented by the circulation of a token.

Figure 2 depicts a superposition step (a *layer* in DisCo parlance) that implements action *delete* atomically. A layer consists of a *requires part* and a *provides part*. The requires part describes the classes and actions that need to be present in a specification in order for the step to be applicable, and the provides part describes the new structure to be added to the specification. New structure is superimposed on classes and actions in *extensions*. The ellipsis “...” in extensions denotes the existing components of the base class or action.

Variables of enumeration types can be used for implementing hierarchical state similar to Statecharts [4]. After declaring variable *status* to be of an enumeration type containing the values *valid* and *invalid*, we can declare variable *next* to be accessible only when the value of *status* is *valid* as follows:

```
status : (valid, invalid);
[status'valid].next : ref node;
```

The requires part describes a closed world, meaning that in a specification to which the step is applied, *delete* must be the only action assigning to the state variable *NEXT*. The closed world assumption along with the initial condition and extensions provided by the layer make it possible to verify temporal properties of the resulting specification.

When verifying temporal properties for a layer we assume that actions in the base specification imply the corresponding actions in the requires part of the layer. When the layer is superimposed on a specification, it is sufficient to verify the action implications to establish that the invariants proved for the layer hold in the result. This is similar to what Katz suggests in [7].

The invariants provided by layer *messages\_l* are:

$$\forall n \in \text{node} : n.\text{status} = \text{valid} \Rightarrow n.\text{NEXT} = n.[\text{status}'\text{valid}].\text{next} \quad (1)$$

$$\begin{aligned} \forall n \in \text{node}, r \in \text{request} : \\ r.\text{existsAs} = \text{message} \wedge r.[\text{existsAs}'\text{message}].\text{from} = \text{ref}(c) \\ \Rightarrow n.\text{NEXT} = r.[\text{existsAs}'\text{message}].\text{next} \end{aligned} \quad (2)$$

$$\begin{aligned} \forall n \in \text{node}, r \in \text{reply} : \\ r.\text{existsAs} = \text{message} \wedge r.[\text{existsAs}'\text{message}].\text{to} = \text{ref}(c) \\ \Rightarrow n.\text{NEXT} = r.[\text{existsAs}'\text{message}].\text{next} \end{aligned} \quad (3)$$

Layer *messages\_l* is superimposed on specification *list* as

```
specification messages is superimpose(messages_l, list);
```

The second parallel branch of specification describes how a token is used for coordinating deletions. In the interest of brevity, we only outline the layer here. It requires class *node* and action *delete*, and provides classes *freeToken* and *reservedToken* and actions *reserveToken*, *passFreeToken* and *passReservedToken*.

Initially there is a single free token which is passed around with action *passFreeToken*. When an object wants to initiate deletion, it waits for the free token. After acquiring the token, the object consumes it and sends a reserved token in action *reserveToken*. The reserved token is then passed around with action *passReservedToken*. Upon receiving a reserved token, an object may execute *delete*, consume the reserved token and send a free token. The safety property provided by the layer is that exactly one token exists at any time and that actions *reserveToken* and *delete* alternate.

### 3 Composition

In our approach, classes and actions in a common ancestor specification provide the points where separately specified aspects meet. If a class or an action has been extended in parallel refinements,

```

layer messages_l
requires
  class node is NEXT : ref node; end;
  action delete by toDelete, pred : node is
  when true do
    pred.NEXT := _;          -- underscore denotes an arbitrary value
    toDelete.NEXT := _;
  end;
provides
  new class request; new class reply;
  class extension node is ...
    status : (valid, invalid); [status'valid].next : ref node;
  end
  class extension request is ...
    existsAs : (nothing, message);
    [existsAs'message].from : ref node; [existsAs'message].next : ref node;
  end
  class extension reply is ...
    existsAs : (nothing, message);
    [existsAs'message].to : ref node; [existsAs'message].next : ref node;
  end
  initially init is
     $\forall$  n : node; req : request; rep : reply ::
      n.status = valid  $\wedge$  req.existsAs = nothing  $\wedge$  rep.existsAs = nothing;
  new action requestDelete; new action completeDelete;
  action extension requestDelete by ... n : node; r : request is
  when ... n.status = valid  $\wedge$  r.existsAs = nothing do ...
    r.existsAs := message;
    r.[existsAs'message].from := ref(n);
    r.[existsAs'message].next := n.[status'valid].next;
    n.status := invalid;
  end;
  action extension delete by ... req : request; rep : reply is
  when ... pred.status = valid  $\wedge$  rep.existsAs = nothing
     $\wedge$  req.existsAs = message  $\wedge$  req.[existsAs'message].from = ref(toDelete)
  do
    -- 'pred.NEXT refers to the value assigned to pred.NEXT in the base specification
    pred.[status'valid].next := 'pred.NEXT;
    req.existsAs := nothing;
    rep.existsAs := message;
    rep.[existsAs'message].to := ref(toDelete);
    rep.[existsAs'message].next := 'toDelete.NEXT;
  end;
  action extension completeDelete by ... n : node; rep : reply is
  when ... rep.existsAs = message  $\wedge$  rep.[existsAs'message].to = ref(n) do
    n.status := valid;
    n.[status'valid].next := rep.[existsAs'message].next rep.existsAs := nothing;
  end;
end;

```

Figure 2: Message exchange as an Ocsid layer

its structure in a composition of the refinements reflects all the extensions. For a class this means that in addition to the state variables in the common ancestor, it contains all the state variables introduced in the refinements. For an action this means that in addition to the structure of the common ancestor it contains all the new roles introduced in the refinements, the guard is a conjunction of the common guard and the new conjuncts, and the body contains the common assignments and the new assignments.

We also allow merging of independently introduced entities. In our example we want a single message to play the role of a delete request and a reserved token. This is done by merging the two classes and the actions that send the messages:

```
specification protocol is
  compose messages, token;
  class requestAndToken := deleteRequest, reservedToken;
  action requestDeletePassToken := requestDelete, reserveToken;
end;
```

Merging classes *deleteRequest* and *reservedToken* into *requestAndToken* means that the three names all denote the same class, we just happened to use different names for the class in the branches. Merging actions is equivalent to executing the component actions simultaneously.

Compared to programming languages, the absence of control flow in joint action specifications simplifies composition considerably. We only need to indicate to which action the new participants, guard conjuncts and assignments are added. Their relative ordering does not matter.

In some ways our composition is similar to that of Tarr et al [13]. Composition of parallel refinements can be seen as a restricted special case of composition of hyperslices. The reason for the much more stringent restrictions on composition in our approach is the desire to preserve temporal properties in composition, which we believe to be of crucial importance in the design of correct distributed systems.

## 4 Abstract Aspects

While the superposition step in Figure 2 can be understood and verified independently, it is still tightly coupled to the specification in Figure 1 because the names of entities in a layer and in a specification need to match.

Verification of temporal properties is insensitive to the names of classes, state variables and actions. For example, if we systematically replace *NEXT* with *V* in step *messages.l* and in the invariants, the modified invariants trivially hold for the modified superposition step.

This observation allows us to express aspects of collective behavior at a more abstract level. For example, instead of specifying how the specific action *delete* is implemented, we can design and verify a superposition step that specifies how to implement an action *A* where two objects of class *C* atomically modify their instance variables *V* of type *T*. This superposition step can then be instantiated for any desired class, action and type. Verification of the abstract step can be reused by establishing that the specification fulfills the assumptions of the superposition step [9].

## 5 Conclusions

As pointed out by Katz in [8], superposition steps can be used to structure specifications according to features of behavior rather than implementation level components. We advocate the use of superposition for aspect-oriented specification of distributed systems for the following reasons. Superposition is a simple, well-understood mechanism with a sound formal basis. Combined with the joint action style of specification it allows one to specify distributed behavior at a level of abstraction where distributed cooperation is explicitly visible.

Temporal properties are an important correctness criteria in the specification of distributed systems. Superposition steps allow one to express cooperation of objects in such a way that

temporal properties can be verified independently. Verification and design effort is reused when a verified superposition step is used in constructing a specification.

Joint actions make the cooperation in a distributed system explicit. The application level semantics can be given at a high level of abstraction, and archived superposition steps can be used for refining the specification into a form that can be implemented in a distributed fashion. Each archived step encapsulates a particular temporal property, so the modularity of the specification matches the modularity of desired properties.

## References

- [1] The DisCo project WWW page. At <http://disco.cs.tut.fi> on the World Wide Web, 2001.
- [2] 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.
- [3] R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with a centralized control. *Distributed Computing*, (3):73–87, 1989.
- [4] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
- [5] Hannu-Matti Järvinen. *The design of a specification language for reactive systems*. PhD thesis, Tampere University of Technology, 1992.
- [6] 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.
- [7] Shmuel Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, April 1993.
- [8] Shmuel Katz and Joseph Gil. Aspects and superimpositions. Position paper at ECOOP’99 AOP workshop, 1999.
- [9] Pertti Kellomäki. A structural embedding of Ocsid in PVS. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics, TPHOLS2001*, number 2152 in Lecture Notes in Computer Science, pages 281–296. Springer Verlag, 2001.
- [10] Reino Kurki-Suonio and Mika Katara. Logical layers in specifications with distributed objects and real time. *Computer Systems Science & Engineering*, 14(4):217–226, July 1999.
- [11] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer Verlag, 1992.
- [12] 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, New Brunswick, NJ, USA, July/August 1996. Springer Verlag.
- [13] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 107–119. IEEE Computer Society Press / ACM Press, 1999.