Design Templates for Collective Behavior

Pertti Kellomäki[1] and Tommi Mikkonen[2]

[1] Software Systems Laboratory, Tampere University of Technology,
P.O Box 553, FIN-33101 Tampere, Finland
pk@cs.tut.fi
[2] Nokia Mobile Phones, Tieteenkatu 1, FIN-33720 Tampere, Finland
Tommi.Mikkonen@nokia.com

Abstract. While sequential behavior of single objects is fairly well understood, orchestrating the collective behavior emerging from the behaviors of individual objects continues to be a challenging task. This is especially true for distributed reactive systems.
The joint action paradigm is a design methodology that concentrates on the collective behavior of objects. Aspects of collective behavior are gradually introduced in a controlled manner in a specification. This paper presents how such aspects can be archived as generic templates, and instantiated in such a way that formal properties verified for a template become properties of its application. Both design and verification effort are reused when a template is applied.

1 Introduction

The traditional unit of modularity in software construction is a component, which is an open system with an interface and some internal state. A complete system is then composed of components. This approach works well at the implementation level, where the goal is to determine the distribution of code and data on the physical components of the system, and to assign responsibilities to teams and individual programmers. However, it is becoming evident that problems at the specification level call for something that extends across objects and ties them together [7].

The work reported here deals with the specification of *reactive distributed systems*, where the coordination of interactions between objects is of interest. Because of the inherent parallelism of such systems and the resulting complexity of the state space, it is very difficult to ascertain the correctness of a given design. An alternative to component based decomposition is to give functionality in an aspect-oriented fashion [15]. Each aspect specifies a view on the collective behavior of the system, and the complete specification is composed by superimposing the aspects. This enables a specifier to concentrate on the correctness of one aspect at a time.

The aspects often follow common idioms so it is advantageous to be able to archive and reuse them. The work presented here describes how this can be done using DisCo [1,10,14] specification language. We focus on safety properties,

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 277–295, 2000.

specifically invariants. The main contribution of this paper is a method by which a design step introducing an aspect of collective behavior can be archived and applied in such a way that safety properties are transferred to the application.

A design step consists of a description of the context in which the step is applicable, and a description of the additional structure (variables and operations on them) that is superimposed on the context to solve a particular design problem. Design steps are expressed as DisCo specifications, so their safety properties may be verified as for any DisCo specification [12]. Applying a design step incurs proof obligations, but the proof obligations are typically considerably easier to verify than the safety properties obtainable from the design step.

The rest of the paper is organized as follows. Section 2 presents necessary background on action systems and the DisCo language. The main contribution of the paper is in Section 3, which describes how design steps are archived and applied. The formal basis of why safety properties are preserved is described in Section 4, and an extended example is given in Section 5. Related work is reviewed in Section 6, and conclusions are drawn in Section 7.

2 The DisCo Language and Action Systems

In this section we describe the joint action [2,3] approach to specification, the use of superposition, and the specification language DisCo.

2.1 Joint Actions

The formal basis of action systems lies in temporal logic. The specific temporal logic used here is Lamport's Temporal Logic of Actions [16] (TLA). An action system consists of a predicate on the initial state, and a number of state transitions called *actions*. An action is a TLA formula that describes a step in a computation. It gives the conditions under which the step may be taken, and defines the values of state variables after the step.

A *joint action* is a formula for a multi-party operation that describes a synchronization of the *participants* of the action. The states of the objects that do not participate the execution remain unchanged. An action may have *parameters*, which are similar to participants except that they denote values, not objects. 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.

2.2 Superposition

Stepwise refinement in the DisCo approach is based on *superposition*. In superposition, 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 also 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.

A specification may contain explicit nondeterminism in the form of action parameters. By augmenting actions with new constraints on the values the parameters may assume, one can constrain the nondeterminism at a later stage.

2.3 The DisCo Language

The experimental specification language DisCo (Distributed Cooperation) has been designed for the incremental development of specifications using superposition. The language syntactically enforces that only superposition steps are used, and it offers a convenient notation for expressing superposition.

The syntactic unit describing a superposition step in DisCo is a *layer*. A layer may import a number of specifications, whose state spaces and actions are merged. The layer may then refine the imported entities and introduce new ones.

State variables reside in *objects*, which are instances of *classes*. Objects are anonymous in the sense that it is not possible to refer to an object by name in a specification. Universal and existential quantification is used for referring to objects. An action lists the participants and their classes, the guard of the action, and the body consisting of assignments to state variables.

Figure 1 depicts a simple DisCo specification *L1*. It describes a world in which two objects of class C may arbitrarily swap the values of their i attributes. The same figure presents a superposition step *L2* that constrains the *swap* action in such a way that the result is a distributed exchange sort. Objects are also augmented with a counter that counts how many times they have been engaged in a *swap* action. The *extend* clause adds new variables to a class, and the ellipsis "..." in the action refinement denotes parts from the previous layer.

2.4 Tool Support

DisCo specifications have a straightforward operational interpretation, so even very high level specifications can be executed. To enable early validation of specifications, an animation tool [23] has been implemented.

Besides validation, the animation tool can be used for rudimentary state space exploration. More formal verification can be carried out using the PVS [21,20] theorem prover. DisCo specifications can be mapped to PVS theories [12] in order to verify invariant properties.

The current DisCo tools do not support parametric specifications as described in this paper, but we plan to extend them so that parametric specifications could be animated and verified in the same fashion as regular DisCo specifications.

```
layer L1 is
  class C is
   i : integer;
  end;

  action swap(c1, c2 : C) is
  when true do
   c1.i := c2.i;
   c2.i := c1.i
  end;
end;
```

```
layer L2 import L1 is
  extend C by
   left : C;
   swaps : integer := 0;
  end;

  refined swap(...) is
  when ... c2.left = c1
      and c1.i > c2.i do
   ...
   c1.swaps := c1.swaps + 1;
   c2.swaps := c2.swaps + 1;
  end;
end;
```

Fig. 1. A DisCo superposition step

3 Archived Steps

In this section we describe the main contribution of the paper, namely how a superposition step can be archived and applied in such a way that design and verification effort is reused.

An archived step consists of a context layer describing the contextual constraints under which the step is applicable, and a solution layer describing the superposition step that can be taken if the constraints are satisfied. Contextual constraints are given in the form of a pattern of interactions between objects that needs to be present in a specification to which the step is to be applied.

If the pattern of objects and their interactions is present in a specification under construction, then an instance of the solution layer can be superimposed on it. The names of variables in the specification under construction may be different from those in the archived step, so the variables need to be linked together. This is accomplished by making the archived step parametric, and instantiating the parameters with entities from the specification under construction when the step is applied.

We use a simple example of token passing in the following. The example illustrates how collective behavior can be captured in an archived step. Section 5 gives a more extended example of archived steps and illustrates other aspects of the mechanism.

3.1 The Context Layer

The role of the context layer is to specify the behavior assumed when the archived step is applied, described operationally using joint actions. The formal parameters of the context layer denote entities (classes, actions, etc.) that need to be provided when the step is applied. The formal parameters are bound to entities in the specification to which the step is being applied.

In DisCo, parameterization is denoted by enclosing the parameters within square brackets. The formal parameter list of a layer must include all the entities (types, classes, functions, relations, initial conditions, assumptions, assertions and actions) defined in the layer. Each of the class parameters includes a list of variables the class should contain, and each of the action parameters includes a list of participants. The lists must agree with the definitions of the classes and actions in the layer.

A simple example of a context layer is presented in Figure 2. The specification describes a system in which objects perform *Act* operations nondeterministically. In order for a specification to match *TokenContext*, it needs to contain a class corresponding to *Object* and an action corresponding to *Act*.

```
layer TokenContext[
      Object :  class;
      Act(o) :  action]  is

  class Object  is
  end;

  action Act(o : Object)  is
  when true  do
  end;
end;
```

Fig. 2. A context layer

A context layer may not include initial conditions or assertions. This restriction is not essential, but it makes it easier to establish refinement between the context layer of a step and a specification to which the step is being applied. If initial conditions are needed for verification of safety properties, they can be introduced in the solution layer.

3.2 The Solution Layer

The solution layer refines the entities in the context layer and introduces new entities, including initial conditions and assertions. The formal parameter list of the solution layer consists of the new entities in the layer. Since they are to be introduced as new entities in the application, the formal parameters are not bound to entities in the specification under construction. Instead, fresh names are provided for them.

An example of a solution layer that can be superimposed on *TokenContext* is presented in Figure 3. It describes how objects coordinate their *Act* operations by circulating a token that grants permission to perform the operation.
The solution layer introduces a new class *Token*, an initial condition *SingleToken* constraining the number of tokens, and a new action to pass the token around.

```
layer TokenSolution[
 Token(at) :  class; SingleToken : initial condition;
 SingleEnabled : assertion; PassToken(t,o) :  action;
 Act(o,t) :  action]  import TokenContext  is

 class Token  is
  at : Object;
 end;

 initially SingleToken  is ∀ t1, t2 : Token :: t1 = t2;

 assert SingleEnabled  is
  not exists o1, o2 : Object ::
        o1 /= o2 and enabled(Act(o=o1)) and enabled(Act(o=o2));

 action PassToken(t : Token; o : Object)  is
 when true  do
  t.at := o;
 end;

 refined Act(... t : Token)  of Act(...)  is
 when ... t.at = o  do
  ...
 end;
end;
```

Fig. 3. A solution layer

The formal safety property guaranteed by the solution layer is expressed as assertion *SingleEnabled*, where *enabled(Act(o=o1))* denotes $\exists tt : Token :: (Guard(Act))(o1, tt)$, i.e., such a token exists that the guard of Act is true. The assertion is easy to verify for the solution layer.

3.3 Applying a Step

Applying an archived step to a specification S under construction involves providing actual parameters for the formal parameters of the layers. This links entities in S with entities in the archived steps.

The actual parameters for the formal parameters of the context layer are entities drawn from S. For the data related entities (classes, types, relations), there needs to be a one to one correspondence. Each action in the context layer may correspond to zero or more actions in S, so there is a list of actions corresponding to each formal action. For a formal action that is refined by the solution layer, the list should contain exactly one actual action. The actual action to be refined is thus always uniquely determined.

It is simple to check that the actual parameter provided for each data-related formal parameter agrees with the declaration of the formal entity in the context layer. For each action provided as an actual parameter, a proof obligation of refinement is obtained.

Consider the following specification:

```
layer Uncoordinated is
  class Node is
   v : integer;
  end;

  class Medium(1) is
   v : integer;
  end;

  action Send(sender, receiver : Node; m : Medium) is
  when true do
   m.v := sender.v;
   receiver.v := sender.v;
  end;
end;
```

The specification describes a system in which a sender and a receiver can communicate synchronously over a medium shared by all nodes (the parenthesized one indicates that there is exactly one medium). The token passing strategy can be applied to coordinate the use of the medium. Figure 4 presents an instance of the solution layer applicable to *Uncoordinated*.

```
layer Coordinated import Uncoordinated is
  assert
   refines(Uncoordinated,
         TokenContext[Uncoordinated.Node; {Uncoordinated.Send(sender)}])
  instance TokenSolution[Token(at); SingleToken;
                         SingleEnabled; PassToken(t,n); Send(n,t)]
  class Token is
   at : Object;
  end;

  initially SingleToken is ∀ t1, t2 : Token :: t1 = t2;

  invariant SingleEnabled is
   not ∃ o1, o2 : Object :: o1 /= o2
         and enabled(Send(sender=o1))
         and enabled(Send(sender=o2));

  action PassToken(t : Token; n : Node) is
  when true do t.at := n; end;

  refined Send(... t : Token) of Send(...) is
  when ... t.at = n do ... end;
end;
```

Fig. 4. Token passing applied to specification *Uncoordinated*

The *assert refines* clause expresses a proof obligation to show that *Uncoordinated* is a refinement of the given instance of *TokenContext*. There are no requirements for the actual parameter corresponding to class *Object* in the context layer, since the formal class does not have any internal structure. The extended example in Section 5 presents a more complex binding of an actual class to a formal class.

The proof obligation for each formal action–actual action pair is to establish that the guard and body of the actual action imply the guard and body of the instance of the formal. The instance of the formal action is derived by replacing the formal relations, types and classes have been replaced by their actual counterparts, and replacing the names of the participants and parameters of the formal action by the names provided in the binding. If the list of actions provided for a formal action is empty, no proof obligation is obtained (one can always think that a corresponding action exists, but its guard is identically false). Furthermore, the actual actions not paired with formal actions must be stuttering with respect to the variables provided as actual parameters.

The instance of the formal action *act* is

action Send(sender : Node) **is**
when true **do**
end;

and the corresponding proof obligation is

$$(true \wedge m.v' = sender.v \wedge receiver.v' = sender.v) \Rightarrow true, \tag{1}$$

which is trivial to verify (primed variables denote the values of the variables after the execution). Section 5 presents examples of more complicated proof obligations.

Intended invariants of an archived step are given as assertions in the solution layer of the step. The solution layer may also include assumptions that can be used when verifying the assertions. When a solution layer is instantiated, assertions in the solution layer become invariants in its instance, and assumptions become assertions (proof obligations). If an assertion has been verified in the archived step, it is sufficient to verify the refinement discussed above and to prove that the proof obligations hold, in order to establish that the assertion is an invariant of the application of the step. Verifying (1) thus establishes *SingleEnabled* in *Coordinated*.

3.4 Tool Support

The instance in Figure 4 was produced manually, but our intent is to provide a mechanical tool to perform the instantiation of archived steps. Such a tool should make matching a specification under construction with a context layer easy. Much of the matching process could be automated using a unification style approach.

The instance of the solution layer is linked to the archived step by the *refines* and *instance* clauses. This is useful for documentation purposes, since it tells the reader of the specification how the layer was obtained. It can also be used for checking that the instance agrees with the archived step, in order to catch modifications to either the instance or the archived step that would invalidate the invariants verified for the step.

4 Formal Basis

In this section we explain the formal basis of why safety properties verified for an archived step hold when it is applied.

Informally the argumentation proceeds as follows. If we can establish that a specification S to which we wish to apply a step implies an instance of the context of the step, then the conjuncts of the actions of the context are in a sense contained in the conjuncts of the actions of S. Applying the solution layer adds equal conjuncts to both S and the solution layer. Hence anything implied by the specification obtained by superimposing the solution layer on the context is also implied by the specification obtained by superimposing the solution layer on S.

More formally, a superposition step is a function $\mathcal{U}$ that maps specifications to specifications. Consider a design step consisting of a context part L_0 and a superposition step $\mathcal{U}$. Verifying a safety property P for $\mathcal{U}(L_0)$ means verifying

$$\mathcal{U}(L_0) \Rightarrow P. \tag{2}$$

When the step is applied, we create an instance $\widehat{L_0}$ of the context and an instance $\widehat{\mathcal{U}}$ of the superposition step by replacing the formal relations, types and classes with the actual entities provided. Since there is a trivial refinement mapping between $\mathcal{U}(L_0)$ and $\widehat{\mathcal{U}}(\widehat{L_0})$, it is true that that

$$\widehat{\mathcal{U}}(\widehat{L_0}) \Rightarrow \widehat{P}, \tag{3}$$

where $\widehat{P}$ denotes an instance of P.

We want to derive a sufficient condition C so that the safety property P also holds in an application $\widehat{\mathcal{U}}(S)$ of the step to specification S, i.e.

$$C \Rightarrow ((\mathcal{U}(L_0) \Rightarrow P) \Rightarrow (\widehat{\mathcal{U}}(S) \Rightarrow \widehat{P})). \tag{4}$$

We proceed by examining actions $\mathcal{A}_{\widehat{\mathcal{U}}(S)}$ in $\widehat{\mathcal{U}}(S)$ and $\mathcal{A}_{\widehat{\mathcal{U}}(\widehat{L_0})}$ in $\widehat{\mathcal{U}}(\widehat{L_0})$. Each action $\mathcal{A}_{\widehat{\mathcal{U}}(S)}$ is a refinement of some action $\mathcal{A}_S$ in S, which is further bound to an action $\mathcal{A}_{\widehat{L_0}}$ in $\widehat{L_0}$ by the formal parameter list of L_0. The superposition step $\widehat{\mathcal{U}}$ may refine $\mathcal{A}_S$ to a disjunction

$$\mathcal{A}^0_{\widehat{\mathcal{U}}(S)} \vee \cdots \vee \mathcal{A}^k_{\widehat{\mathcal{U}}(S)} \tag{5}$$

and $\mathcal{A}_{\widehat{L}_0}$ to

$$\mathcal{A}^0_{\widehat{\mathcal{U}}(\widehat{L}_0)} \vee \cdots \vee \mathcal{A}^k_{\widehat{\mathcal{U}}(\widehat{L}_0)}, \tag{6}$$

but there is a one to one correspondence between the disjuncts (a superposition step always introduces a fixed number of refinements). It is thus sufficient to consider a single pair of actions $\mathcal{A}_{\widehat{\mathcal{U}}(S)}$ and $\mathcal{A}_{\widehat{\mathcal{U}}(\widehat{L}_0)}$.

Let $\mathcal{A}_{\widehat{L}_0}$ be defined as

$$\mathcal{A}_{\widehat{L}_0} = \exists p_1, \ldots, p_n : F_{\widehat{L}_0}(p_1, \ldots, p_n). \tag{7}$$

Since a superposition step cannot remove conjuncts from an action, its refinement is of the form

$$\begin{aligned} \mathcal{A}_{\widehat{\mathcal{U}}(\widehat{L}_0)} = \exists & p_1, \ldots, p_n, q_1, \ldots, q_m : \\ & F_{\widehat{L}_0}(p_1, \ldots, p_n) \\ & \wedge F^{new}_{\widehat{\mathcal{U}}}(p_1, \ldots, q_m). \end{aligned} \tag{8}$$

Similarly, let $\mathcal{A}_S$ be defined as

$$\mathcal{A}_S = \exists p_1, \ldots, p_l : F_S(p_1, \ldots, p_l). \tag{9}$$

Applying $\widehat{\mathcal{U}}$ to it adds the same participants and parameters $q_1, \ldots, q_m$, and conjuncts $F^{new}_{\widehat{\mathcal{U}}}$:

$$\begin{aligned} \mathcal{A}_{\widehat{\mathcal{U}}(S)} = \exists & p_1, \ldots, p_l, q_1, \ldots, q_m : \\ & F_S(p_1, \ldots, p_l) \\ & \wedge F^{new}_{\widehat{\mathcal{U}}}(p_1, \ldots, q_m) \end{aligned} \tag{10}$$

If we assume

$$F_S(p_1, \ldots, p_l) \Rightarrow F_{\widehat{L}_0}(p_1, \ldots, p_n), \tag{11}$$

it is possible to write $\mathcal{A}_S$ equivalently as

$$\begin{aligned} \mathcal{A}_{\widehat{\mathcal{U}}(S)} = \exists & p_1, \ldots, p_l, q_1, \ldots, q_m : \\ & F_{\widehat{L}_0}(p_1, \ldots, p_n) \\ & \wedge F_S(p_1, \ldots, p_l) \\ & \wedge F^{new}_{\widehat{\mathcal{U}}}(p_1, \ldots, q_m) \end{aligned} \tag{12}$$

This is a simple consequence of the tautology $(Q \Rightarrow R) \Rightarrow (Q \Leftrightarrow Q \wedge R)$. From (12), and (8) it follows that

$$\mathcal{A}_{\widehat{\mathcal{U}}(S)} \Rightarrow \mathcal{A}_{\widehat{\mathcal{U}}(\widehat{L}_0))}. \tag{13}$$

The reasoning above applies to all actions in $\widehat{\mathcal{U}}(S)$, since each of them is a refinement of some action in S and thus a refinement of some action in $\widehat{L}_0$. It follows that if $\widehat{P}$ is a safety property of $\widehat{\mathcal{U}}(\widehat{L}_0)$, no action in $\widehat{\mathcal{U}}(S)$ can invalidate it:

$$(\widehat{\mathcal{U}}(\widehat{L}_0) \Rightarrow \widehat{P}) \Rightarrow (\widehat{P} \wedge \mathcal{A}_{\widehat{\mathcal{U}}(S)} \Rightarrow \widehat{P}') \tag{14}$$

L_0 does not contain any initial conditions, so the initial conditions of $\widehat{\mathcal{U}}(\widehat{L_0})$ are by construction present in $\widehat{\mathcal{U}}(S)$ as well, as they are introduced by $\widehat{\mathcal{U}}$. We thus conclude that (11) is a sufficient condition for the preservation of safety properties of $\mathcal{U}(L_0)$ in $\widehat{\mathcal{U}}(S)$.

5 An Extended Example

In this section we give an extended example of an archived step that illustrates some aspects not present in the token passing example. The example deals with the common problem of communicating some data over a channel that cannot directly represent the data. At the sending end a datum to send is decomposed into lower level data, and at the receiving end it is composed back to a higher level datum. The obvious safety property is that the lower level channel should implement a higher level channel.

Figure 5 presents a parameterized DisCo layer describing the contextual constraints of the decomposition–composition step. This time the context is somewhat more complicated, involving variables and state changes. The specification describes a system in which data is produced to the sending buffer of the sender, sent to the receiver, and consumed from the receiving buffer of the receiver.

Besides the entities introduced in *DecomposeComposeContext*, its formal parameters include the type *lowT* and the functions *decompose* and *compose*, which are not used in the layer. The reason for their appearance is that they are entities used in the solution layer that should nevertheless be provided when the step is applied.

Figure 6 depicts the solution part of the step. In the solution, a datum to be sent is first decomposed into lower level date, and the lower level data are sent one by one to the receiver. Transmitting the final low level datum corresponds to the high level send action,

The invariant ensured by the solution is *lowBufferComposition*, which states that the composition of the low level data in the *lowBuffer* variables equals the first item to be sent in the sender's high level sending queue. In particular, this means that a receiver does not need access to the sender's high level sending queue in action *sendLast*. It is sufficient to access the local low level receiving queue. The specification does not specify the mechanism by which the receiver recognizes the last low level datum, since there are several possibilities a particular application could use.

The invariant can be verified using the initial conditions *SingleSender* and *SingleReceiver* which constrain the number of objects, and the assumption *IsInverse* which describes the properties required from the functions.

Figure 7 presents a specification to which we wish to apply the decomposition – composition step. It describes a weather sensor that measures temperature and humidity and sends the measurements to a recorder that stores the sequence of measurements. Actions *senseTemperature* and *senseHumidity* model the independent sensing of the environment, and action *measure* models reading the sensors and storing the measurement to be transmitted.

```
layer DecomposeComposeContext[
 Sender(dataToSend), Receiver(receivedData) : class;
 highT, lowT : type; decompose, compose : function;
 produce(s,d), send(s,r), consume(r) : action] is

  class Sender is
   dataToSend : sequence hightT;
  end;

  class Receiver is
   receivedData : sequence hightT;
  end;

  action produce(s : Sender; d : highT) is
  when true do
   s.dataToSend := s.dataToSend & <d>;
  end;

  action send(s : Sender; r : Receiver) is
  when size(s.dataToSend) > 0 do
   r.receivedData := r.receivedData & <head(s.dataToSend)>;
   s.dataToSend := tail(s.dataToSend);
  end;

  action consume(r : Receiver) is
  when size(r.receivedData) > 0 do
   r.receivedData := tail(r.receivedData);
  end;
end;
```

Fig. 5. Context part of decomposition–composition

The specification does not include functions corresponding to *compose* and *decompose*, so they are introduced in a separate layer *WeatherFunctions* (Figure 8) before applying the step. Action *measure* is also augmented with a parameter matching d in *produce*.

Applying the step to *WeatherFunctions* yields an instance of *DecomposeComposeSolution*. Figure 9 presents an excerpt of the instance. Actions have been omitted, as they are simply derived by making the substitutions of actual parameters to formal parameters.

Compared to the token passing example, an application of the decompose–compose step incurs more complicated proof obligations. The proof obligations related to action *produce* in the archived step is

```
layer DecomposeComposeSolution[
 Sender(lowBuffer), Receiver(lowBuffer) : class;
 emptyBuffers, SingleSender, SingleReceiver : initial condition;
 IsInverse : assumption; lowBufferComposition : assertion;
 prepareSend(s),lowSend(s,r),sendLast(s,r) : action]
import DecomposeComposeContext is
  extend Sender by lowBuffer : sequence lowT; end;

  extend Receiver by lowBuffer : sequence lowT; end;

  initially emptyBuffers is
   ∀ s : Sender; r : Receiver :: s.lowBuffer = < > and r.lowBuffer = < >;

  initially SingleSender is ∀ s1, s2 : Sender :: s1 = s2;
  initially SingleReceiver is ∀ r1, r2 : Receiver :: r1 = r2;

  assumption IsInverse is
    ∀ ht : highT :: compose(decompose(ht)) = ht;

  assert lowBufferComposition is
   ∀ s : Sender; r : Receiver ::
     size(r.lowBuffer & s.lowBuffer) /= 0
     => compose(r.lowBuffer & s.lowBuffer) = head(s.dataToSend);

  action prepareSend(s : Sender) is
  when size(s.lowBuffer) = 0 and size(s.dataToSend) > 0 do
   s.lowBuffer := decompose(head(s.dataToSend));
  end;

  action lowSend(s : Sender; r : receiver) is
  when size(s.lowBuffer) > 1 do
   r.lowBuffer := r.lowBuffer & <head(s.lowBuffer)>;
   s.lowBuffer := tail(s.lowBuffer);
  end;

  refined sendLast(...) of send(...) is
  when ... size(s.lowBuffer) = 1 do
   ...
   s.lowBuffer := < >;
   r.lowBuffer := < >;
  end;
end;
```

Fig. 6. Solution part of decomposition–composition

$$
\begin{aligned}
&(size(s.sendingBuffer) = 0 \\
&\wedge m = Measurement(s.temperatureReading, s.humidityReading) \\
&\wedge s.sendingBuffer' = \\
&\qquad \langle Measurement(s.temperatureReading, s.humidityReading)\rangle) \\
\Rightarrow \\
&(true \wedge s.sendingBuffer' = s.sendingBuffer \,\&\, \langle m\rangle),
\end{aligned}
\tag{15}
$$

```
layer WeatherSensor is
  type Measurement is
   temperature : real;
   humidity : real;
  end;

  class Sensor is
   temperatureReading : real;
   humidityReading : real;
   sendingBuffer : sequence Measurement;
  end;

  class Recorder is measurements : sequence Measurement; end;

  action senseTemperature(s : sensor; t : real) is
  when true do
   s.temperatureReading := t;
  end;

  action senseHumidity(s : sensor; h : real) is
  when true do
   s.humidityReadon := h;
  end;

  action measure(s : Sensor) is
  when size(s.sendingBuffer) = 0 do
   s.sendingBuffer := <Measurement(s.temperatureReading,
                                   s.humidityReading)>;
  end;

  action transmit(s : Sensor; r : receiver) is
  when size(s.sendingBuffer) = 1 do
   r.measurements := r.measurements & <head(s.sendingBuffer)>;
   s.sendingBuffer := < >;
  end;
end;
```

Fig. 7. Specification of a weather station

which is relatively straightforward to verify. The proof obligation related to action *send* is

$$
\begin{aligned}
&(size(s.sendingBuffer) = 1 \\
&\wedge r.measurements' = r.measurements \,\&\, \langle head(s.sendingBuffer)\rangle \\
&\wedge s.sendingBuffer' = \langle\rangle) \\
\Rightarrow \quad & \\
&(size(s.sendingBuffer) > 0 \\
&\wedge r.measurements' = r.measurements \,\&\, \langle head(s.sendingBuffer)\rangle \\
&\wedge s.sendingBuffer' = tail(s.sendingBuffer)).
\end{aligned}
\tag{16}
$$

```
layer WeatherFunctions import WeatherSensor is
  function MeasurementToReals(m : Measurement) : sequence real is
   <m.temperature, m.humidity>
  end;

  function RealsToMeasurement(s : sequence real) : Measurement is
   Measurement(head(s), head(tail(s)))
  end;

  refined measure(... m : Measurement) of measure(...) is
  when ... m = Measurement(s.temperatureReading, s.humidityReading)
  do ... end;
end;
```

Fig. 8. Adapting *WeatherSensor* to *DecomposeComposeContext*

This is also easy to verify. As there is no action in *WeatherSensor* corresponding to *consume*, no proof obligations arise.

The assumption *IsInverse* in the solution layer gives rise to a further proof obligation as assertion *IsInverse* in *WeatherSolution*. Verifying it is also a simple exercise in reasoning about records and sequences.

6 Related Work

Formal refinement of specifications into executable programs has been extensively studied (see e.g. [4,19]). Our work is different from the mainstream refinement research because of the use of superposition. A superposition step is a refinement step by construction, so refinement need not be verified. Instead, we are interested in safety properties related to the new variables introduced in a superposition step. These properties may link newly introduced variables to the higher level variables (data refinement), or they may be e.g. consistency properties of the new variables.

Compared to refinement of specifications into sequential programs, our work has somewhat different focus. Instead of algorithmic complexity, we are interested in managing the complexity emerging from interactions of objects. In this context, complex local computations can often be abstracted away using nondeterminism, and sophisticated data refinement is not necessary. For a fully formal derivation of distributed systems the sequential computations embodied in actions would also need to be derived, but this is not our present goal.

Archiving design steps is well established in the context of sequential programs. The Specware [11] system and others developed in the Kestrel Institute have been notably successful in the derivation of executable programs from algebraic specifications [22,5]. In Specware, refinement is based on *interpretations*. An interpretation $\mathcal{A} \to S$ is a pair of *morphisms* $S \to S'$ and $\mathcal{A} \to S'$, where $\mathcal{S}'$ is a definitional extension of S. In our construction $S \to S'$ is the superposition relation, and $\mathcal{A} \to S'$ is the composition of instantiation and refinement.

```
layer WeatherSolution  import WeatherFunctions
assert
 refines(WeatherFunctions,
      DecomposeComposeContext[WeatherFunctions.Sensor(sendingBuffer),
           WeatherFunctions.Recorder(measurements),
           WeatherFunctions.Measurement, real,
           WeatherFunctions.MeasurementToReals,
           WeatherFunctions.RealsToMeasurement,
           {WeatherFunctions.measure(s,d)},
           {WeatherFunctions.transmit(s,r)}, {}]
instance DecomposeComposeSolution[emptyBuffers, SingleSensor,
        SingleRecorder, IsInverse, lowBufferComposition, Sensor(lowBuffer),
          Recorder(lowBuffer), prepareSend(s), lowSend(s,r), sendLast(s,r)]
is
 extend Sensor  by lowBuffer : sequence real;  end;
 extend Recorder  by lowBuffer : sequence real;  end;

 initially emptyBuffers  is
 ∀ s : Sensor; r : Recorder :: s.lowBuffer = < > and r.lowBuffer = < >;

 initially SingleSensor  is ∀ s1, s2 : Sensor :: s1 = s2;
 initially SingleRecorder  is ∀ r1, r2 : Recorder :: r1 = r2;

 assert IsInverse  is
 ∀ ht : Measurement ::
     RealsToMeasurement(MeasurementToReals(ht)) = ht;

 invariant lowBufferComposition  is
 ∀ s : Sensor; r : Recorder ::
   r.lowBuffer & s.lowBuffer /= < > =>
  RealsToMeasurement(r.lowBuffer & s.lowBuffer) = head(s.sendingBuffer);

          ⋮
end;
```

Fig. 9. Application of *DecomposeComposeSolution* to *WeatherFunctions*

By restricting to the special case, we manage without the rather heavy mathematical machinery underneath Specware. Avoiding mathematics is of course not a value in itself, but our approach meshes well with how development is normally done in DisCo. It also uses concepts that are close to those of programming, which has been a design goal when developing the DisCo approach.

Our work is also connected with *patterns* [8,6]. A specification in the closed world approach embodies a solution and the context in which the solution is applicable. Together with an informal problem statement, these meet a common definition of what constitutes a pattern. A joint action specification formally describes relations and interactions between objects, which in pattern literature are usually given informally using class diagrams accompanied with prose and descriptions of message interchange. Behavioral patterns can easily be expressed

as closed world joint action specifications. Work on formalizing patterns in the joint action formalism has been reported in [18].

Compositional TLA (cTLA) [9] is an approach closely related to DisCo, as both are specification languages based on Temporal Logic of Actions [16]. The languages support and encourage quite different styles of specification, though. Compared to DisCo, cTLA has a more component oriented flavor: a specification is composed of *processes*, which are subsystems described in TLA. Each process has a set of private state variables, which are only accessible to the actions of the process. When a new process is composed, actions of the component processes are synchronized, and action parameters are used for sharing values among the synchronized actions.

The cTLA *pattern integration* [17] operation can be used for merging the state spaces of component processes. Integration is similar to composition, but additionally one gives a new set of state variables. The state variables of the components are replaced with expressions involving the new state variables. The result is a process that has the properties of its components, but a flat rather than a hierarchical structure. The pattern integration operation can be used to achieve results similar to those of our approach, but with a considerably different style.

At a first glance, archived joint actions specifications may resemble e.g. Ada generic packages or C++ templates. There is an important difference brought about by the use of superposition, however. Instead of components, our archived units describe *aspects* of behavior. In the programming language world the closest analogy is *aspect oriented programming* [13], where fragments of code called aspects are woven into a complete program by an aspect weaver. The ellipsis "..." in DisCo plays the role of the aspect weaver.

7 Conclusions and Further Work

We have presented how aspects of collective behavior can be archived and applied as parametric joint action specifications. Applying such archived aspects facilitates the reuse of both design and verification effort. Safety properties may be verified for an archived specification. These properties are transferred to an application of the specification by discharging a set of proof obligations that are typically easier to verify than the safety property itself.

Further work is needed for finding aspects that are worthwile to archive and verify. In connection with the DisCo method we see the greatest potential in situations where atomicity needs to be ensured e.g. by some form of locking. Implementing data abstractions is another possible area of application. We plan to extend the current DisCo tools so that parametric specifications could be animated and verified.

Acknowledgments. This work has been partly funded by the Academy of Finland, project 757473.

References

1. The DisCo project WWW page. At URL http://disco.cs.tut.fi/ on the World Wide Web, 1999.
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. Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.
5. Lee Blaine, Li-Mei Gilham, Junbo Liu, Douglas R. Smith, and Stephen Westfold. Planware – domain-specific synthesis of high-performance scheduler. In *Proceedings of the Thirteenth Automated Software Engineering Conference*, pages 270–280. IEEE Computer Society Press, 1998.
6. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns*. John Wiley & Sons, 1996.
7. J. O. Coplien. Idioms and patterns as architectural literature. *IEEE Software*, pages 36–42, January 1997.
8. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.
9. P. Herrmann, G. Graw, and H. Krumm. Compositional specification and structured verification of hybrid systems in cTLA. In *Proceedings of the 1st IEEE International Symposium on Object-oriented Real-time distributed Computing - (ISORC'98)*, pages 335–340. IEEE Computer Society Press, 1998.
10. 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.
11. R. Juellig, Y. Srinivas, and J. Liu. SPECWARE: An advanced environment for the formal development of complex software systems. Lecture Notes in Computer Science, Vol. 1101, Springer–Verlag, 1996.
12. Pertti Kellomäki. Verification of reactive systems using DisCo and PVS. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*. Lecture Notes in Computer Science, Vol. 1313, pages 589–604. Springer–Verlag, 1997.
13. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference. Lecture Notes in Computer Science*, Vol. 1241, pages 220–242. Springer–Verlag, 1997
14. 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.
15. Reino Kurki-Suonio and Tommi Mikkonen. Abstractions of distributed cooperation, their refinement and implementation. In B. Krämer, N. Uchihira, P. Croll, and S. Russo, editors, *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 94–102. IEEE Computer Society, April 1998.
16. Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

17. Arnulf Mester and Heiko Krumm. Formal behavioural patterns for the tool-assisted design of distributed applications. In Hartmut König, Kurt Geihs, and Thomas Preuß, editors, *IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS 97)*, pages 235–248. Chapman & Hall, 1997.
18. Tommi Mikkonen. Formalizing design patterns. In B. Werner, editor, *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pages 115–124. IEEE Computer Society, April 1998.
19. Carroll Morgan. *Programming from Specifications: Second Edition.* Prentice Hall International, Hempstead, UK, 1994.
20. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction.* Lecture Notes in Artificial Intelligence, Vol. 607, pages 748–752. Springer–Verlag, 1992.
21. Sam Owre, John Rushby, Natrajan Shankar, and Friedrich von Henke. Formal verification of fault-tolerant architecures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
22. D. Smith and E. Parra. Transformational approach to transportation scheduling. In *Proceedings of the 8th Knowledge-Based Software Engineering Conference*, pages 60–68. IEEE Computer Society Press, September 1993.
23. Kari 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.