

A Layer-Based Formalization of an On-Board Instrument

Tommi Mikkonen
Software Systems Laboratory
Tampere University of Technology
P.O. Box 553, FIN-33101 Tampere, Finland
Email: tjm@cs.tut.fi

Abstract

Logical layers enable structuring of reactive systems in a fashion that is orthogonal to conventional components. We introduce a case study, where such layers are used to structure a formal specification of an on-board instrument. The specification starts at an abstract level, and advances towards a detailed description of the instrument with refinement steps whose size is manageable. Finally, we present a summary on the experiences obtained when composing the specification.

1 Introduction

Logical layers enable the use of collaboration as criteria for modularity. Such layers have been conventionally used for decomposition of existing systems, for instance, in program slicing [13, 14] and projections [8] used in verification, but they can also be used for composing new specifications. With logical layers, the focus of each design step taken when deriving a specification is explicitly set on the cooperation of components constituting the system. Moreover, layers provide an architecture orthogonal to that of conventional component-based approaches [11, 10], where each layer introduces one view or aspect with respect to interparty cooperation.

In this paper, we present a layer-based specification of COALA [15], an on-board ozone measuring instrument intended to be attached on an earth-orbiting spacecraft. The formalization relying on logical layers is presented with the *DisCo* method intended for modeling of reactive systems, i.e., those that are in continuous interaction with their environments.

The rest of the paper is structured as follows. Section 2 introduces the formalism used in the specification, and Section 3 provides an insight on the instrument formalized in this paper. Sections 4, 5, and 6 form the core of this paper, and provide a layer-based specification of COALA, with focus on collaboration of different entities. Finally, Section 7 summarizes the paper.

2 The DisCo Method

The DisCo¹ method [1, 2, 6] is a formal specification method intended for the specification of reactive systems. Semantics of the method have been defined with the Temporal Logic of Actions, TLA [9], and many of the details of the method are similar to those used in UNITY [3]. The purpose of this section is to introduce basics of the method, and the language used for composing specifications [5].

2.1 Classes, Objects, and Actions

DisCo specifications are introduced as *systems* that are defined in terms of *classes* and *actions*. Classes are patterns for *objects* that can contain local variables or (hierarchical) statemachines. Classes are defined in the format

```
class class name is
  internals
end class name;
```

¹ Acronym for *Distributed Cooperation*.

```

system SimpleSort is

  class Process is
    state *booting, ready;
    extend ready by value : integer; end ready;
    next : Process;
  end Process;

  assert MutuallyExclusiveStates is  $\forall p : \text{Process} :: p.\text{booting} \vee p.\text{ready}$ ;

  action Initialize(i : integer) by *p : Process is
  when p.booting do
     $\rightarrow$  p.ready;
    p.ready.value := i;
  end Initialize;

  action Exchange(i, j: integer) by p1, p2 : Process is
  when p1.next = p2  $\wedge$  i = p1.ready.value  $\wedge$  j = p2.ready.value  $\wedge$  i > j do
    p1.ready.value := j;
    p2.ready.value := i;
  end Exchange;
end SimpleSort;

```

Figure 1: A simple DisCo specification

Internals of a class consist of pairs of variable names and their types, or of statemachines whose default states are denoted with asterisks. Values of all variables (and current states of the associated statemachines) in one object constitute the *local* state of the object. The *global* state of a system is the combination of all local states.

Actions are multi-object operations that modify local states of involved objects. They are given in the format

```

action action name by participants is
when guard do
  statements
end action name;

```

In an action, participants are pairs consisting of a *role name* and a class. Guard is a boolean expression made of variables of objects taking the roles. The guard determines when the action can be executed. Statements formalize the effect of an execution of the action. State hierarchies are handled by interpreting any reference to a variable internal to a state to imply that the object is in the associated state when evaluating guards, and by requiring that all accesses in the body are made to an active state. Asterisks are used to attach fairness requirements to participant objects. Such requirements ensure that if an object can take certain role in the execution of an action infinitely often, the action will be executed for the object repeatedly.

In addition to classes and actions, a DisCo specification can introduce invariants. They are given as *assertions*,

```

assert assertion name is boolean expression;

```

Assertions can be used to formalize the intended topology of a specification, or as a watchdog for a certain undesired situation when animating the specification with the associated animation tool [12]. Obviously, assertions also enable rigorous reasoning, with the option to use a theorem prover [7].

Use of these constructs is exemplified in Figure 1, which introduces a simple DisCo specification. The specification defines how distributed processes forming an array sort their values in to an increasing order, assuming that their topology is such that processors indicate the subsequent process with variable *next*. As states of a statemachine are always disjoint, the assertion given in the figure is superfluous, included in the specification only to demonstrate the use of assertions.

2.2 Property-Preserving Layers

DisCo specifications can be made more precise by refining them. The method allows refinements that apply *superposition*, a well-known mechanism first used in [4]. With such refinements, new classes can be introduced, and those that already exist can be extended. For actions, it is required that refined actions always logically imply the ones they have been derived from. This ensures that *safety properties*² are preserved by construction. For *liveness properties*³, additional considerations are required.

In practice, classes can be extended with the following format:

```
extend class name by  
  extensions  
end class name;
```

where *extensions* refer to new variables or statemachines, introduced similarly to those in new classes. Extensions can also be addressed to individual states, thus enabling state hierarchies.

Actions are refined in the format

```
refined new action name (... new parameters) of old action name  
by ... new participants is  
when ... new conjuncts do  
  new statements  
  ...  
  more new statements  
end new action name;
```

In the action, each ellipsis (...) represents the corresponding part of the original action. One action can be given several refinements in one layer, and each action can be refined in several layers. If the name and parameter and participant lists are not modified, corresponding references to the ancestor action can be omitted from the refinement, resulting in the following shorthand:

```
refined action name is  
when ... new conjuncts do  
  new statements  
  ...  
  more new statements  
end action name;
```

Use of refinements is exemplified in Figure 2, where a simple DisCo refinement has been given. The specification refines previously introduced DisCo system into a form where variables are used to indicate potential directions for further exchange operations.

2.3 Inheritance

A simple form of inheritance, reminding aggregation rather than inheritance in the conventional object-oriented sense, is included in the DisCo method. This allows objects of a subclass to inherit the ability to participate in actions in the roles of their superclasses. The use of inheritance is introduced in the format:

```
class subclass name is  
  inherit superclass name as new variable name;  
end subclass name;
```

Multiple inheritance is allowed.

When using inheritance, actions can be specialized for participants in certain subclasses. This is expressed in the format

²Intuitively, safety properties are of the form “Something bad will never happen.”

³Intuitively, liveness properties are of the form “Something good will eventually happen.”

```

system RefinedSort import SimpleSort; is

  extend Process.ready by left, right : boolean; end Process.ready;

  refined Exchange is
  when ... p1.ready.right  $\vee$  p2.ready.left do
    ...
    p1.ready.right := false;
    p1.ready.left := true;
    p2.ready.right := true;
    p2.ready.left := false;
  end Exchange;

  action Agree by p1, p2 : Process is
  when p1.next = p2
     $\wedge$  p1.ready.value  $\leq$   $\wedge$  j = p2.ready.value
     $\wedge$  (p1.ready.right  $\vee$  p2.ready.left)
  do
    p1.ready.right := false;
    p2.ready.left := false;
  end Agree;
end RefinedSort;

```

Figure 2: A simple DisCo refinement

```

specialized new action name (... new parameters) of old action name
by ... new participants, object relation is
when ... new conjuncts do
  new statements
  ...
  more new statements
end new action name;

```

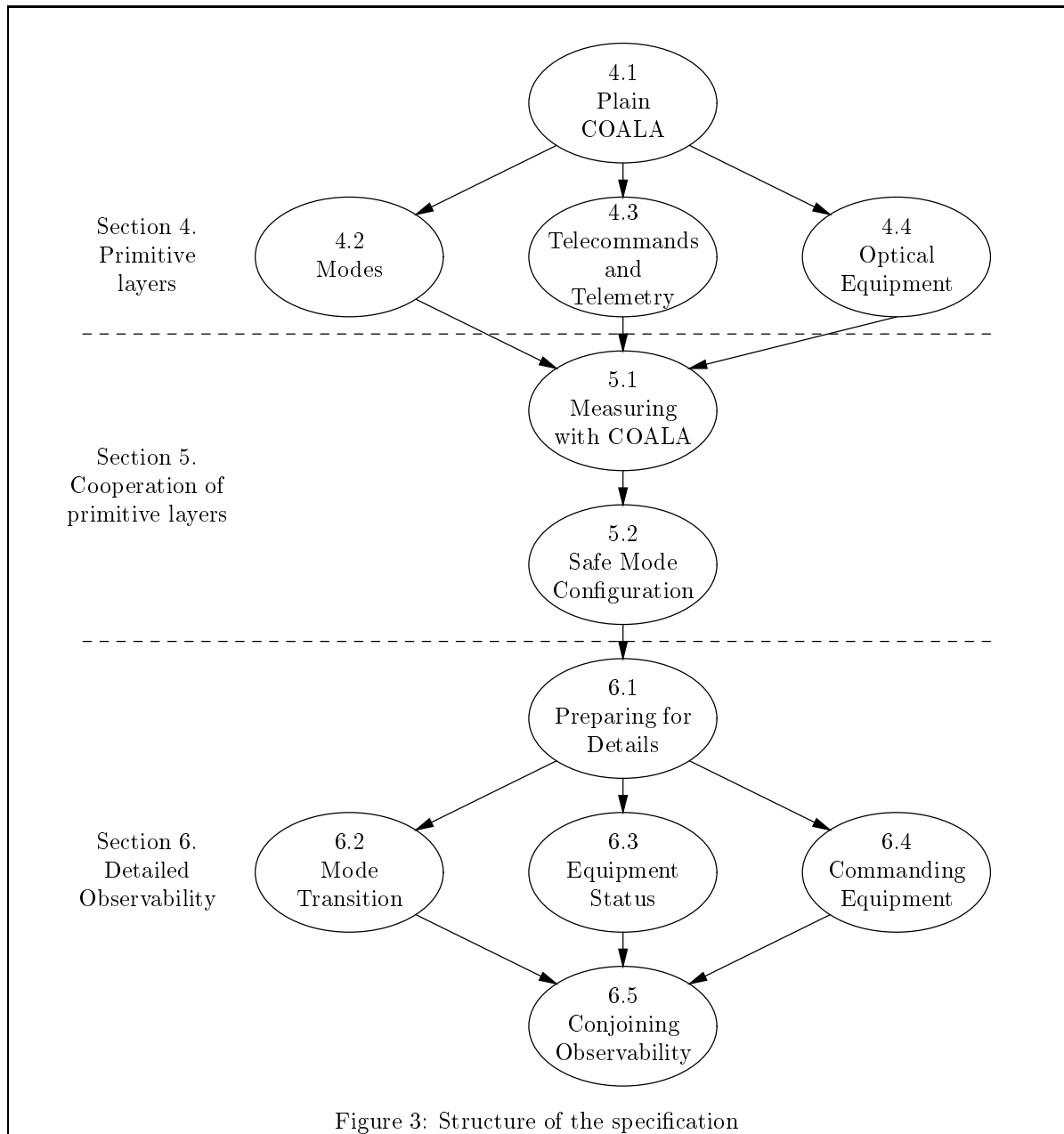
where *object relation* defines for which subclass the specialization is given. The original action can no longer be used with subclasses involved in the specialization, but it is still available for the rest of the subclasses.

3 Introduction to the On-Board Instrument

COALA (Calibration for Ozone through Atmospheric Limb Acquisitions) is a future on-board instrument designed for ozone measurements [15]. The aim of the COALA study has been to identify ozone measuring instrument concepts that would represent interesting complements or alternatives for mission scenarios being planned or assessed for the next 10 to 15 years. Foreseen mission scenarios include validation, calibration, and complementary measurements when cooperating with another instrument, and long-term trend measurements when functioning as a stand-alone system.

The relation of different layers used in the formalization is illustrated in Figure 3. In the figure, names of subsections introducing DisCo specifications are used to refer to layers. In addition to individual layers constituting the specification, the figure indicates different sections discussing formalization with dashed lines.

This formalization stops at the level of telecommands and telemetry associated with them. They are introduced in different layers in order to enable focusing on one telecommand at a time, and the final layer conjoins specifications introducing them. In this case, arising from the viewpoint of a feasibility study [15], we consider that telecommands and telemetry are an adequate abstraction on what the instrument will perform. Obviously, when design advances, there can be lower-level concepts, including, for instance, processes, subsystems, and pieces of equipment. The lowest level can then



```

system PlainCoala is
  class Coala is end;
end PlainCoala;

```

Figure 4: Simplest version of COALA

be seen as a description of an individual process, subsystem, or a piece of an equipment described in connection with its environment.

4 Primitive Layers

This section introduces the fundamental formalization of class *Coala* to be extended in subsequent layers, gives a specification of the mode of the instrument, defines how the instrument communicates with the outside world, and introduces optical equipment needed in mission requirements. The section introduces these aspects with four DisCo layers each of which is discussed in a separate subsection.

4.1 Plain COALA

As indicated in Figure 3, different branches of the formalization share an ancestor specification. This ancestor is given in Figure 4. The specification introduces an empty class *Coala*, which is extended in different layers. The specification is needed, as otherwise there would be no possibility to use class *Coala* in all branches, because a straightforward introduction of class *Coala* in each branch would lead to a name conflict when conjoining them. An alternative solution would be to use multiple inheritance, but this would result in a superfluously complex specification due to the restricted view DisCo has on inheritance.

In practice, the need for trivial layers introducing only empty classes, which are then to be extended in subsequent layers, is a fairly common situation when different views to a system are independently formalized.

4.2 Modes

A DisCo specification that introduces the mode of the instrument is provided in Figure 5. The mode is defined as a state hierarchy, where there are three top-level states, *launch*, *run*, and *eol*. Of these states, *launch* and *eol* model inactivity in different phases of the mission, and *run* is the state that represents activity. State transitions between these states are referred to as *Start* and *EndOfLife* in the associated specification, respectively.

State *run* is extended with four states that represent the operational mode of the instrument. When the instrument is turned on, it enters *commission* mode, in which initial switch-on is performed. When all pieces of equipment have been validated to function normally, the instrument can be set to *operation* mode, where nominal activities of the instrument are performed. If a failure occurs in *operation* mode, the instrument is set to *safe* mode, in which COALA is reconfigured into a configuration in which no failures are foreseen. When the instrument has recovered from the failure, it can be set to *standby* mode, in which only nominal pieces of electronics are available for testing and validation purposes. The mode can be left for *commission* or *operation* mode. Actions that cause the discussed mode transitions are referred to as *EnterCommission*, *EnterOperations*, *EnterSafe*, and *EnterStandby* in the associated specification.

The instrument has two operational activities. When performing *search* activity, the instrument searches a target star, and when performing *measure* activity, the instrument takes measurements on ozone. Action *ChangeActivity* changes the activity being currently performed. Details of activities mostly consist of algorithmic computations. In this formalization, such computations will be omitted due to focusing on the collective behavior of different parties constituting the instrument. If necessary, an independent layer could be given, that would formalize the computation, with the option to extend states modeling the activities.

```

system Modes import PlainCoala; is

  extend Coala by
    state *launch, run, eol;
    extend run by
      state *commission, operation, safe, standby;
      extend operation by state *search, measure; end operation;
    end run;
  end Coala;

  action Start by c : Coala is
  when c.launch do
    → c.run.commission;
  end Start;

  action EndOfLife by c : Coala is
  when ¬c.eol do
    → c.eol;
  end EndOfLife;

  action EnterCommission by c : Coala is
  when c.run.standby do
    → c.run.commission;
  end EnterCommission;

  action EnterOperations by c : Coala is
  when c.run.commission ∨ c.run.standby do
    → c.run.operation;
  end EnterOperations;

  action EnterSafe by c : Coala is
  when c.run.operation do
    → c.run.safe;
  end EnterSafe;

  action EnterStandby by c : Coala is
  when c.run.safe do
    → c.run.standby;
  end EnterStandby;

  action ChangeActivity by c : Coala is
  when c.run.operation do
    if (c.run.operation.search) then → c.run.operation.measure;
    else → c.run.operation.search;
    end if ;
  end ChangeActivity;
end Modes;

```

Figure 5: Introducing modes

```

system TCTM import PlainCoala; is

  class Telecommand is state *newTC, executingTC, deletedTC; end TeleCommand;

  class Telemetry is state *newTM, generatedTM, deletedTM; end Telemetry;

  extend Coala by
    tc : Telecommand;
    tm : Telemetry;
  end Coala;

  action ReceiveTC by tc : Telecommand; c : Coala is
  when c.tc = null  $\wedge$  tc.newTC do
    c.tc := tc;
     $\rightarrow$  tc.executingTC;
  end ReceiveTC;

  action CompleteTC by tc : Telecommand; tm : Telemetry; c : Coala is
  when c.tc = tc  $\wedge$  tc.executingTC  $\wedge$  c.tm = null  $\wedge$  tm.newTM do
    c.tc := null;
     $\rightarrow$  tc.deletedTC;
    c.tm := tm;
     $\rightarrow$  tm.generatedTM;
  end CompleteTC;

  action SendTM by tm : Telemetry; c : Coala is
  when c.tm = tm  $\wedge$  tm.generatedTM do
    c.tm := null;
     $\rightarrow$  tm.deletedTM;
  end SendTM;
end TCTM;

```

Figure 6: Telecommanding and Telemetry

4.3 Telecommands and Telemetry

A DisCo specification introducing a simple form of telecommanding and telemetry is given in 6. At this point of the development, we pay no attention to the contents of the messages, but place the focus on the execution scheme. We introduce a generic telecommand, and define how the instrument responds to the reception of the telecommand in terms of telemetry. Detailed descriptions on how telecommands affect the instrument will be discussed in Section 6.

The specification models telecommands and telemetry messages as classes whose internal state-machine models the dynamic existence of such messages. In a telecommand (telemetry message), state *newTC* (*newTM*) models those telecommands (telemetry messages) that have not yet been sent, and state *deletedTC* (*deletedTM*) represents telecommands (telemetry messages) that have already been executed. State *executingTC* (*executingTM*) is a telecommand (telemetry message) that is currently being processed (generated) by the instrument.

Executions of telecommands and generation of telemetry are synchronized, because all telecommanding and associated telemetry transmissions take place via an OBDH bus that connects COALA with the hosting spacecraft. Therefore, even if some data is acquired from the instrument, a telecommand is assumed to be used to acquire data. Each execution takes place in a fashion where a telecommand is first received with action *ReceiveTC*. Then, the instrument executes the telecommand and generates associated telemetry by executing action *CompleteTC*. Finally, generated telemetry is transmitted by executing action *SendTM*.

4.4 Optical Equipment

Optical equipment is introduced in the specification given in Figure 7. The structure of the specification is the following. Optical equipment consists of a mirror that is used for searching a target star, and measurements with respect to ozone. The equipment can be turned on and off, modeled with the obvious statemachine, thus enabling optimization on power usage. Moving of the mirror is modeled with two integer variables, *pointingAzim* and *pointingElev*. Minimum and maximum values of these variables have been obtained by dividing the pointing range with associated pointing accuracy.

In addition to the equipment used for measurements, the specification defines how acquired measurement data is downlinked. Downlinking of science data is introduced in a fashion reminding that used for ordinary telemetry messages, i.e., statemachine is again used to model dynamic existence of objects. The difference, however, is that here, the instrument is responsible for generating and downlinking of telemetry with no stimuli from outside environment. This is possible, as science data is transmitted via special science interface unit only used for this purpose. Science data is included in the specification as an integer introduced in class *ScienceTM*. From the viewpoint of logic, there is no indication of size for an integer, and therefore the variable can be understood to represent arbitrary amount of information.

Actions are given for turning optics on and off, formalized as *TurnOn* and *TurnOff*, respectively. Action *MoveMirror* is used for moving the mirror. In the action, the new position of the mirror is indicated by parameters *i* and *j* that are required to be near enough to current position. Generation and downlinking of science data is handled with actions *GenerateSC* and *SendSC*. In action *GenerateSC*, parameter *d* represents the data to be downlinked. At this level of abstraction, this value is nondeterministic, but a more elaborate formalization could define how the value is acquired from associated electronic equipment. If desired, a new layer could be used to formalize such aspects.

5 Cooperation of Primitive Layers

In this section, we conjoin the layers discussed in the previous section. The logic of the specification obtained from this conjunction is introduced in two phases. The first phase introduces refinements that ensure that the instrument is active whenever it is performing something. The second phase defines *safe* mode reconfiguration for optical equipment, which consists of turning the mirror into a harmless direction and switching off the optical equipment.

5.1 Measuring with COALA

COALA capable of measuring is obtained when layers introducing mode, telecommands and telemetry, and the optical instrument are conjoined. This conjunction has been formalized in the specification provided in Figure 8.

Essential refinements with respect to optical equipment enforce that the mirror can be moved (action *MoveMirror*), the optics switched on (action *TurnOn*), and science telemetry generated (action *GenerateSC*) and downlinked (action *SendSC*) only when the instrument is active, i.e., the instance of class *Coala* is in state *run*. In addition, science telemetry can only be generated in *operation* and *commission* modes, which reflect the intended usage of the instrument. For telecommanding and telemetry, the specification introduces refinements that strengthen the guards of actions *ReceiveTC*, *CompleteTC*, and *SendTM* to enforce that telecommands and telemetry can only be processed when the instrument is functioning.

Fairness requirements are introduced that force the instrument to generate and transmit scientific telemetry, and to process telecommands and generate telemetry when the instrument is active. The requirements are given with respect to role *c* given for an object of class *Coala* to indicate that the instrument is the active component of this specification. The actions with respect to whom fairness requirements are given are such that they are executed autonomously by the instrument. No indication of liveness is given for actions representing the activity of outside world.

5.2 Safe Mode Configuration

Specification introducing *safe* mode configuration is given in Figure 9, where class *Optics* is extended with values representing the safe position of the mirror. In addition, actions controlling optics are re-

```

system Optics import plainCoala; is

  class Optics is
    state on, *off;
    pointingAzim, pointingElev : integer;
  end Optics;

  class ScienceTM is state *newSC, generatedSC(d : integer), deletedSC; end ScienceTM;

  extend Coala by
    op : Optics;
    sc : ScienceTM;
  end Coala;

  action TurnOn by o : Optics; c : Coala is
  when o.off  $\wedge$  c.op = o do
     $\rightarrow$  o.on;
  end TurnOn;

  action TurnOff by o : Optics; c : Coala is
  when o.on  $\wedge$  c.op = o do
     $\rightarrow$  o.off;
  end TurnOff;

  action MoveMirror(x, y : integer) by o : Optics; c : Coala is
  when o.on
     $\wedge$  c.op = o
     $\wedge$  (o.pointingAzim - 1  $\leq$  x  $\leq$  o.pointingAzim + 1)
     $\wedge$  (0  $\leq$  x  $\leq$  42000)
     $\wedge$  (o.pointingElev - 1  $\leq$  y  $\leq$  o.pointingElev + 1)
     $\wedge$  (0  $\leq$  y  $\leq$  1750)
  do
    o.pointingAzim := x;
    o.pointingElev := y;
  end moveMirror;

  action GenerateSC(d : integer) by sc : ScienceTM; c : Coala; o : Optics is
  when o.on  $\wedge$  c.op = o  $\wedge$  c.sc = null  $\wedge$  sc.newSC do
    c.sc := sc;
     $\rightarrow$  sc.generatedSC(d);
  end GenerateSC;

  action SendSC by sc : ScienceTM; c : Coala is
  when c.sc = sc  $\wedge$  sc.generatedSC do
    c.sc := null;
     $\rightarrow$  sc.deletedSC;
  end SendSC;
end Optics;

```

Figure 7: Optical equipment

```

system Measuring import Modes, TCTM, Optics; is

  refined TurnOn is
  when ... c.run do
    ...
  end TurnOn;

  refined MoveMirror is
  when ... c.run do
    ...
  end MoveMirror;

  refined GenerateSC by ... *c is
  when ... c.run.operation  $\vee$  c.run.commission do
    ...
  end GenerateSC;

  refined SendSC by ... *c is
  when ... c.run do
    ...
  end SendSC;

  refined ReceiveTC is
  when ... c.run do
    ...
  end ReceiveTC;

  refined CompleteTC by ... *c is
  when ... c.run do
    ...
  end CompleteTC;

  refined SendTM by ... *c is
  when ... c.run do
    ...
  end SendTM;
end Measuring;

```

Figure 8: Measuring with COALA

```

system SafeConf import Measuring; is

  extend Optics by
    safeAzim, safeElev : integer;
  end Optics;

  function CalculateSafeDirection(i, j : integer) return integer is
    if i > j then 1
    elsif i < j then -1
    else 0
    end if ;
  end CalculateSafeDirection;

  refined SafeOff of TurnOff is
  when ... c.run.safe  $\wedge$  o.safeAzim = o.pointingAzim  $\wedge$  o.safeElev = o.pointingElev do
    ...
  end SafeOff;

  refined ManualTurnOff of TurnOff is
  when ... do
    ...
  end ManualTurnOff;

  refined MoveToSafe of MoveMirror is
  when ... c.run.safe
     $\wedge$  (o.safeAzim  $\neq$  o.pointingAzim  $\vee$  o.safeElev  $\neq$  o.pointingElev)
     $\wedge$  x = CalculateSafeDirection(o.safeAzim, o.pointingAzim)
     $\wedge$  y = CalculateSafeDirection(o.safeElev, o.pointingElev)
  do
    ...
  end MoveToSafe;

  refined MoveMirror is
  when ...  $\neg$ c.run.safe do
    ...
  end MoveMirror;
end SafeConf;

```

Figure 9: Safe mode reconfiguration

financed twice in order to explicitly indicate the difference between an autonomous event and an operation initiated by the hosting spacecraft.

In more detail, actions *TurnOff* and *MoveMirror* are refined into actions *SafeOff* and *ManualTurnOff*, and *MoveToSafe* and *MoveMirror*, respectively. Of them, actions *SafeOff* and *MoveToSafe* represent autonomous operations, and actions *ManualTurnOff* and *MoveMirror* are responses to events initiated by the hosting spacecraft (or a user commanding the spacecraft). The latter actions are required, as otherwise actions *SafeOff* and *MoveToSafe* would be the only refinements of actions *TurnOff* and *MoveMirror* in this layer, invalidating the options to turn off optics and control the mirror in any other way. Action *ManualTurnOff* is available even if a reconfiguration is currently being performed. Reconfiguration taking place in *safe* mode can thus be interrupted by turning off optical equipment manually.

6 Detailed Observability

This section introduces detailed descriptions of three types of telecommands and telemetry necessary for describing the behavior of the instrument in more detail. In the following subsection, we provide a specification that introduces all telecommands and telemetry messages discussed in detail.

```

system PrepareForDetails import SafeConf; is

  class ToSafeTC is
    inherit TeleCommand as TC;
  end ToSafeTC;

  class ToSafeTM is
    inherit Telemetry as TM;
  end ToSafeTM;

  class StatusTC is
    inherit TeleCommand as TC;
  end StatusTC;

  class StatusTM is
    inherit Telemetry as TM;
  end StatusTM;

  class TurnOffTC is
    inherit TeleCommand as TC;
  end TurnOffTC;

  class TurnOffTM is
    inherit Telemetry as TM;
  end TurnOffTM;
end PrepareForDetails;

```

Figure 10: Telecommands and telemetry to be represented in detail

Then, each type of a telecommand and associated telemetry, including mode transition, status query, and commanding of a piece of an equipment, is discussed in the following subsections. Finally, the telecommands are conjoined into one specification, which then represents the final version of COALA.

6.1 Preparing for Details

As already discussed, it is often necessary to introduce empty classes when preparing for independent refinements. Here, we plan to introduce different telecommands in parallel branches of the specification. However, in order to keep the relation between different types of telecommands clear, we introduce all the telecommands in a separate layer. This layer is provided in Figure 10.

In the formalization, classes representing telecommand and telemetry associated with entering *safe* mode, acquiring information on the position of the mirror, and turning the optical equipment off are formalized by deriving new subclasses from classes *Telecommand* and *Telemetry*. This results in special versions of telecommands and telemetry messages that are used in these contexts only.

6.2 Mode Transition

Specification introducing a mode transition telecommand is given in Figure 11. The specification extends classes *ToSafeTC* and *ToSafeTM* with statemachines needed for modeling processing of such telecommands and generation of associated telemetry messages. More precisely, when an instance of class *ToSafeTC* is in state *idle*, it can be immediately be received. In state *pend*, the command is being processed by the instrument. States *ok* and *error* model a telecommand after a successful execution or an occurrence of an error. For telemetry, state *idle* is a new telemetry message, and *ok* and *error* are states that indicate whether the associated telecommand was successful.

Executions of telecommands that set the instrument to *safe* mode and generation of associated telemetry takes place as follows. First, the instrument receives a telecommand by executing *ReceiveSafeTC*. Then, the instrument enters *safe* mode by executing action *SafeByTC*, or acknowledges that the telecommand is invalid by executing action *SafeTCError*. In any case, associated telemetry is

```

system ToSafeTCTM import PrepareForDetails; is

  extend ToSafeTC by state *idle, pend, ok, error; end ToSafeTC;

  extend ToSafeTM by state *idle, ok, error; end ToSafeTM;

  specialized ReceiveSafeTC of ReceiveTC by ... tstc : ToSafeTC, tc = tstc.TC is
  when ... tstc.idle do
    ...
    → tstc.pend;
  end ReceiveSafeTC;

  refined SafeByTC of enterSafe by ... *c; tstc : ToSafeTC is
  when ... tstc.TC = c.tc ∧ tstc.pend do
    ...
    → tstc.ok;
  end SafeByTC;

  action SafeTCError
  by tstc : ToSafeTC; *c : Coala is
  when tstc.TC = c.tc ∧ tstc.pend ∧ c.run ∧ ¬c.run.operation do
    → tstc.error;
  end SafeTCError;

  specialized CompleteSafeByTC of CompleteTC
  by ... tstc : ToSafeTC, tc = tstc.TC; tstm : ToSafeTM, tm = tstm.TM is
  when ... (tstc.ok ∨ tstc.error) ∧ tstm.idle do
    ...
    if (tstc.ok) then → tstm.ok;
    else → tstm.error;
    end if ;
  end CompleteSafeByTC;
end ToSafeTCTM;

```

Figure 11: Telecommand and telemetry for setting the instrument to Safe mode

```

system StatusTCTM import PrepareForDetails; is

  extend StatusTC by state *idle, pend; end StatusTC;

  extend StatusTM by
    state *idle, ok, error;
    azim, elev : integer;
  end StatusTM;

  specialized ReceiveStatusTC of ReceiveTC by ... stc : StatusTC, tc = stc.TC is
  when ... stc.idle do
    ...
    → stc.pend;
  end ReceiveStatusTC;

  specialized CompleteStatusTC of CompleteTC
  by ... stc: StatusTC, tc = stc.TC; stm: StatusTM, tm = stm.TM; o : optics is
  when ... stc.pend  $\wedge$  stm.idle  $\wedge$  o = c.op do
    ...
    if (o.on) then stm.azim := o.pointingAzim; stm.elev := o.pointingElev; → stm.ok;
    else → stm.error;
    end if ;
  end CompleteStatusTC;
end StatusTCTM;

```

Figure 12: Telecommand and telemetry for reading mirror position

generated after processing the telecommand by executing action *CompleteSafeByTC*, with a convention that indicates successful and erroneous executions in the resulting telemetry. Further refinements could be given to include more information on potential errors in telemetry.

Fairness requirements are given with respect to processing of the telecommand either as a successful mode change, or as an error notification. No new fairness requirements are necessary for action *CompleteSafeByTC*, because associated fairness requirements have already been given in previous layers.

6.3 Equipment Status

Specification introducing a telecommand that acquires equipment status is given in Figure 12. The specification uses the telecommand reading the position of the mirror as an example. Associated telemetry message therefore contains data fields used for downlinking mirror position.

Formalization is based on a strategy similar to that used in mode transition. However, as the reception of the telecommand has no effect on the internal state of the instrument, execution can be simplified, resulting in a scheme where a telecommand is first received when action *ReceiveStatusTC* is executed, and associated telemetry can be generated immediately with action *CompleteStatusTC*. Action *CompleteStatusTC* takes care for generating both success and failure messages, depending on the current state of the instrument. There is no need to give new fairness requirements, because all necessary fairness requirements have already been given in the layer that conjoins the mode and observability.

6.4 Commanding Equipment

Specification introducing a telecommand used to access a piece of an equipment is given in Figure 13. The specification uses the telecommand that turns off optical equipment as an example.

Formalization is based on a strategy similar to that used in mode transition. Telecommands turning pieces of equipment on and off are received with action *ReceiveTurnOffTC*. Then, two versions are provided for an execution of the telecommand, action *TurnOffByTC* that represent successful

```

system TurnOffTCTM import PrepareForDetails; is

  extend TurnOffTC by state *idle, pend, ok, error; end TurnOffTC;

  extend TurnOffTM by state *idle, ok, error; end TurnOffTM;

  specialized ReceiveTurnOffTC of ReceiveTC by ... totc : TurnOffTC, tc = totc.TC is
  when ... totc.idle do
    ...
    → totc.pend;
  end ReceiveTurnOffTC;

  refined TurnOffByTC of ManualTurnOff by ... *c; totc : TurnOffTC is
  when ... totc.TC = c.tc ∧ totc.pend do
    ...
    → totc.ok;
  end TurnOffByTC;

  action TurnOffError
  by totc : TurnOffTC; *c : Coala is
  when totc.TC = c.tc ∧ totc.pend ∧ c.run ∧ c.op.off do
    → totc.error;
  end TurnOffError;

  specialized CompleteTurnOffTC of CompleteTC
  by ... totc : TurnOffTC, tc = totc.TC; totm : TurnOffTM, tm = totm.TM is
  when ... (totc.ok ∨ totc.error) ∧ totm.idle do
    ...
    if (totc.ok) then → totm.ok;
    else → totm.error;
    end if ;
  end CompleteTurnOffTC;
end TurnOffTCTM;

```

Figure 13: Telecommand and telemetry for turning off optical equipment

execution, and action *TurnOffError* modeling an error taking place in the execution. In this context, the only error that can take place is that the optical equipment is already turned off.

6.5 Conjoining Observability

Final version of COALA is obtained when layers introducing telecommands and associated telemetry are conjoined. The specification is listed in Figure 14. Conjunction of branches introducing detailed telecommands and telemetry requires no refinement to classes or actions.

Obviously, there are more telecommands and telemetry messages than discussed in this section, but they can be formalized similarly to the ones given above. Augmentation of new telecommands and associated telemetry would imply changes to system *PrepareForDetails*, and introduction of new parallel branches to define the behavior related to them. However, this task falls beyond the scope of this paper.

7 Summary

We provided a layer-based specification of COALA, an on-board instrument used for ozone measurements. The case study demonstrates that complexity of inter-component cooperation in real systems can be managed with logical layers. In particular, the fact that the size of individual steps is manageable is remarkable. Further, the development initiated at an abstract level can be refined into a form where the effect of individual telecommands can be identified in their natural contexts, with the

```
system DetailedTCTM
  import ToSafeTCTM, StatusTCTM, TurnOffTCTM;
is
end DetailedTCTM;
```

Figure 14: Final version of COALA

option to consider the context in abstract or in detailed form. Due to the use of property-preserving refinements, abstractions introduced in the very first layers are guaranteed to be valid at the level of abstraction that discusses telecommands related to a certain functionality. This obviously supports tracing between different versions of the specification, thus increasing the confidence on the quality of the resulting specification.

Liveness properties were incorporated in the specification at a relatively late stage. We find that the possibility to postpone their introduction is essential for layerwise specification, as fairness requirements should only be introduced when essential safety properties have been specified. This is not in line with the philosophy adopted in UNITY [3], for instance, where uniform fairness requirements are given with respect to all operations, disregarding the level of abstraction of the specification. This results in problems, when properties are abstracted away in early phases of specification.

Total amount of work invested in this specification is about 1.5 man-months. This includes the time it took to get to know COALA, the time spent on composing numerous versions of the specification, and restructuring resulting in intuitively well-justified layers. A part of the time has been spent on finding refinement steps simple enough to fit on one page of a document, although this is a requirement arising from practical and typographical rather than strictly technical needs. Writing of draft versions of this report is not included in the period, although experiences obtained with them have had a tremendous impact on the layers used in the final specification.

Acknowledgments

A simplified version of this paper has been included in [10], where the focus is to demonstrate different mechanisms that are necessary for layer-based specification.

I wish to express my sincere thanks to Space Systems Finland for providing the opportunity to study COALA with DisCo.

References

- [1] The DisCo project WWW page. At URL <http://www.cs.tut.fi/ohj/DisCo/> on the World Wide Web.
- [2] R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with a centralized control. *Distributed Computing*, (3):73–78, 1989.
- [3] K. M. Chandy and J. Misra. *Parallel Program Design, A Foundation*. Addison-Wesley, 1988.
- [4] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.
- [5] H.-M. Järvinen. *The design of a specification language for reactive systems*. PhD thesis, Tampere University of Technology, 1992.
- [6] 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.
- [7] P. Kellomäki. Verification of reactive systems using DisCo and PVS. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, number 1313 in Lecture Notes in Computer Science, pages 589–604. Springer-Verlag, 1997.

- [8] S. S. Lam and A. U. Shankar. Protocol verification via projections. *IEEE Transactions on Software Engineering*, 10(4):325–342, 1984.
- [9] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
- [10] T. Mikkonen. Abstractions and logical layers in specifications of interactive systems. PhD thesis, to appear.
- [11] T. Mikkonen. The two dimensions of an architecture. Accepted to First Working IFIP Conference on Software Architecture, to appear.
- [12] K. Systä. A graphical tool for specification of reactive systems. In *Proceedings of the Euromicro'91 Workshop on Real-Time Systems*, pages 12–19, Paris, France, June 1991. IEEE Computer Society Press.
- [13] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.
- [14] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.
- [15] Coala final report. COA-VTT-TN-008, ESTEC Contract Number 11714/95/NL/CN, December 1996.