On the Horizontal Dimension of Software Architecture in Formal Specifications of Reactive Systems

Mika Katara
Institute of Software Systems
Tampere University of Technology
P.O. Box 553, FIN-33101
Tampere, Finland
mika.katara@tut.fi

Reino Kurki-Suonio
Institute of Software Systems
Tampere University of Technology
P.O. Box 553, FIN-33101
Tampere, Finland
reino.kurki-suonio@tut.fi

Tommi Mikkonen
Institute of Software Systems
Tampere University of Technology
P.O. Box 553, FIN-33101
Tampere, Finland
tommi.mikkonen@tut.fi

ABSTRACT

In order to provide better alignment between conceptual requirements and aspect-oriented implementations, formal specification methods should enable the encapsulation of *logical abstractions* of systems. In this paper we argue that *horizontal architectures*, consisting of such logical abstractions, can provide better separation of concerns over conventional ones while supporting incremental development for more common units of modularity such as classes. We base our arguments on our experiences with the DisCo method, where logical abstractions are composed using the *superposition* principle.

1. INTRODUCTION

Post-object programming (POP) mechanisms, like those developed in aspect-oriented programming [6], provide means to modularize crosscutting concerns, which are in some sense orthogonal to conventional modularity. The background of this paper is in the observation that the same goal has been pursued also at the level of formal specifications of reactive systems, and that the results of this research are relevant for the theoretical understanding of POP-related architectures and of the associated specification and design methods.

Unlike conventional software modules, units of modularity that are suited for a structured description of the intended logical meaning of a system can be understood as aspects in the sense of aspect-oriented programming. We call such units *horizontal* in contrast to conventional *vertical* units of modularity, such as classes and processes. While the vertical dimension remains dominant because of the available implementation techniques, the horizontal dimension can provide better separation of concerns over the vertical one improving, for example, traceability of requirements.

In this paper, our experiences with the DisCo method are used as the basis for discussion. The rest of the paper is structured as follows. First, in Section 2, we present the idea of structuring specifications using horizontal units capturing logical rather than structural abstractions of the system. In Section 3 the DisCo method is presented which utilizes such components as primary units of modularity. Section 4 concludes the paper by discussing the approach in the light of related work.

2. TWO DIMENSIONS OF SOFTWARE ARCHITECTURE

Describing an architecture means construction of an abstract *model* that exhibits certain kinds of intended properties. In the following we consider *operational* models, which formalize executions as state sequences, as illustrated in Figure 1, where all variables in the model have unique values in each state s_i. In algorithmic models these state sequences are finite, whereas in reactive models they are nonterminating, in general. Message sequence charts are a well-known operational formalism for describing state sequences where states s_i consist only of the control points of the communicating processes.

2.1 Vertical Units

The algorithmic meaning of software, as formalized by Dijkstra [4], has the desirable property that it can be composed in a natural manner from the meanings of the components in a conventional architecture. To see what this means in terms of executions in operational models, consider state sequences that implement a required predicate transformation. Independently of the design principles applied, a conventional architecture imposes a "vertical" slicing on these sequences, so that each unit is responsible for certain *subsequences* of states. This is illustrated in Figure 2, where the satisfaction of the precondition-postcondition pair (P, Q) for the whole sequence relies on the assumption that a subsequence V, generated by an architectural unit, satisfies its precondition-postcondition pair (P_V, Q_V).

More generally, an architecture that consists of conventional units imposes a nested structure of such vertical slices on each state sequence. In the generation of these sequences, the two basic operations between architectural units can

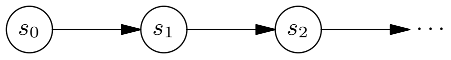

Figure 1: Execution as a state sequence.

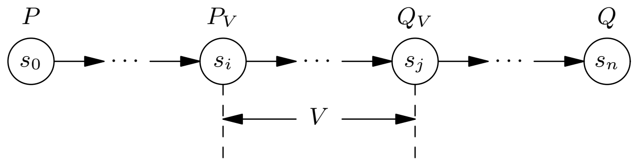

Figure 2: A vertical slice V in an execution.

be characterized as *sequential composition* and *invocation*. The former concatenates state sequences generated by component units; the latter embeds in longer sequences some state sequences that are generated by a component unit. In both cases, the resulting state sequences have *subsequences* for which the components are responsible. In current software engineering approaches, this view has been adopted as the basis for designing behaviors of object-oriented systems, leading the focus to interface operations that are to be invoked, and to the associated local precondition-postcondition pairs.

The architectural dimension represented by this kind of modularity will be called *vertical* in the following.

2.2 Horizontal Units

The meaning of a system can also be modeled by how the values of its variables, denoted by set X, behave in nonterminating state sequences. In order to have modularity that is natural for such a reactive meaning, the meanings of the components must be of the same form. In other words, each component must also generate nonterminating state sequences, but the associated set of variables can be a subset of X. An architecture of reactive units therefore imposes a "horizontal" slicing of state sequences, so that each unit is responsible for some subset X_H of variables in all states s_i, as illustrated in Figure 3.

In the generation of state sequences, only one basic operation is needed. *Superposition* uses state sequences that are generated by a horizontal slice embedding them in sequences that involve a larger set of variables. The state sequences of the resulting vertical architecture have *projections* for which the horizontal components are responsible. Properties of horizontal slices then emphasize collaboration between different vertical units, and the relationships between their internal states.

The two dimensions of architecture are in some sense dual to each other. On the one hand, from the viewpoint of vertical architecture, the behaviors generated by horizontal units represent crosscutting concerns. From the horizontal viewpoint, on the other hand, vertical units emerge incrementally.

2.3 Architecting Horizontal Abstractions

To illustrate the nature of horizontal units, consider a simple modeling example of an idealized doctors' office, where

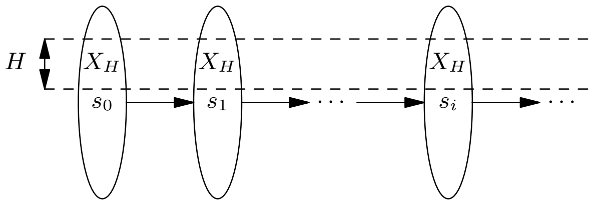

Figure 3: A horizontal slice H in an execution.

ill patients are healed by doctors.[1] The natural vertical units in such a model would include patients, doctors and receptionists. Horizontal units, on the other hand, would model their cooperation as specific projections of the total system, and the whole specification could be built incrementally from these.

The specification process can start with a trivial model of the simple aspect that people get ill, and ill patients eventually get well. The "illness bits" of the patients are the only variables that are needed in this horizontal unit. Next, this unit can be embedded in a larger model where a patient gets well only when healed by a doctor. This extended model has events where a doctor starts inspecting a patient, and participation of a doctor is also added to the events where a patient gets well. Finally, a further superposition step can add the aspect that also receptionists are needed in the model, to organize patients to meet doctors, and to make sure that they pay their bills. This aspect is truly crosscutting in the sense that it affects all the vertical units, i.e., patients, doctors and receptionists.

Each unit in this kind of a horizontal architecture is an *abstraction of the meaning* of the total system. The first horizontal unit in this example is an abstraction where all other behavioral properties have been abstracted away except those that concern the "illness bits" of patients. In terms of Temporal Logic of Actions (TLA) [18], (the meaning of) the total system always implies (the meaning of) each horizontal unit in it. As for variables, each component in the horizontal structure focuses on some variables that will become "secrets" encapsulated in the vertical components in an eventual implementation. This can be related with the observation of [19], where such secrets are considered more important than the interfaces associated with them in early phases of design.

This gives a formal basis for specifying a reactive system – i.e., for expressing its intended meaning – incrementally in terms of operational abstractions that can be formally reasoned about. Since it is unrealistic to formulate any complex specification in one piece, this is a major advantage for using horizontal architectures in the specification process. A classical example of using horizontal slices is the separation of correctness and termination detection in a distributed computation [5]. This is also the earliest known use of superposition in the literature – its close relationship with aspect-orientation was first reported in [13].

For comparison, consider how stepwise refinement proceeds

[1]This is an outline of a simplified version of an example that was used to illustrate the ideas of DisCo in [16].

with vertical architectural units. In terms of executions, each operational abstraction generates state sequences that lead from an initial state to a final state, so that the required precondition-postcondition pairs are satisfied. At the highest level of abstraction there may be only one state change, and each refinement step replaces some state changes by state sequences that are generated by more refined architectural units. This leads to a design where the early focus is on interfaces and their use, whereas the "secrets" inside the vertical components may become available only towards the end of the design, when the level of implementable interface operations is achieved.

Due to the above, the abstractions that a vertical architecture provides are not abstractions of the meaning: the complete meaning is assumed to be available already at the highest level, and it remains the same throughout the design process. Instead, at each level of refinement, a vertical architecture gives an *abstraction of the structure* of an implementable operational model.

3. EXPERIENCES WITH DISCO

The above views have been stimulated by the experiences gained with the DisCo[2] method [10, 23]. DisCo is a formal specification method for reactive system, whose semantics are in TLA [18].

3.1 Horizontal Architectures in DisCo

In DisCo, the horizontal dimension, as discussed above, is used as the primary dimension for modularity. The internal structure of horizontal units consists of partial classes that reflect the vertical dimension. For instance, each of the attributes of a class can be introduced in different horizontal units.

Behavioral modeling is DisCo's bread and butter. The design usually advances so that first the high-level behaviors are included in the model. Based on this abstract behavioral model it is then possible to include more details, even to the level where direct mapping to available implementation techniques becomes an option [17].

In more detail, horizontal components correspond to superposition steps referred to as *layers*. Formally, each layer is a mapping from a more abstract vertical architecture to a more detailed one. As the design decisions are encapsulated inside the layers, they become first-class design elements. Because layers represent logical, rather than structural aspects of the system, they serve in capturing concepts of the problem domain.

Ideally, each layer contains only those details that pertain to a particular logical aspect. Thus, better alignment between the requirements and design can be achieved. In cases where a layered structure is aligned with an aspect-oriented implementation [1], this results in improved traceability concerning the entire design flow.

Dependencies between layers arise if, for instance, a class is defined in one layer and its attribute in another layer. In this case the latter is said to depend on the former. These

[2]Acronym for Distributed Co-operation.

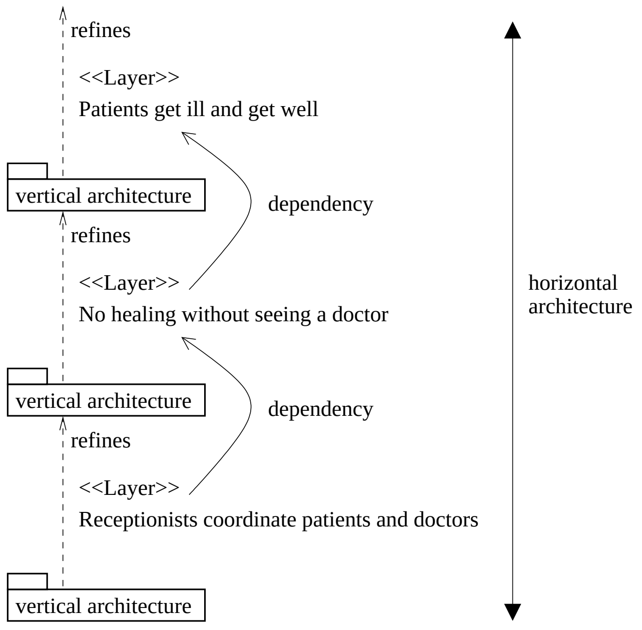

Figure 4: Horizontal architecture consisting of layers.

dependencies impose a partial order between the layers of the specification.

In Figure 4 the situation is illustrated in the simple case of the example outlined above. The horizontal architecture consists of three layers. Each layer refines a vertical architecture to a more detailed one by adding a new piece of "meaning" to the system. The layer introducing doctors depends on the one introducing patients, and the one introducing receptionists depends on the one introducing doctors (and transitively on the patients layer).

Different concerns can be partially treated in common layers. A situation of this kind arises when a common design decision is made to cater for two different features, for instance. This means that, generally, concerns are treated in one or more layers. Moreover, there can be overlap between the sets of layers treating different concerns.

3.2 Example: Mobile Robot

As a more concrete example, a simplified DisCo specification of a mobile robot (toy car) control software specification is presented. We have omitted the parts dealing with real time (and fairness) which can be found in [11] in a slightly different form.

The mobile robot is a small microcontroller-based car. The objective is to keep the car on a track marked by optical tape. From the viewpoint of the control software the system has two inputs and two outputs. The inputs are readings from an A/D converter connected to infra-red censors, and from an odometer. The outputs are two servo motors controlling the steering and the movement. The servos are driven by PWM (Pulse Width Modulation) signals. There is also a switch, which is used to start the car and to stop it.

There are two concern that need to be addressed: the basic functionality of the car including starting and stopping and the control part including the control algorithms. These concerns are treated in three separate layers, one of which is common to both concerns, i.e. the concerns are overlapping. The details of the layers are described next.

3.2.1 Basic Actions

Layer Basic_Actions contains the basic functionality of the system. It introduces two *classes* and three *multi-object actions.* Actions are symmetric with respect to participants; there are no callers nor callees. Class Data holds internal *variables* and class Output variables that model the outputs. The variables r_dist and r_tape of type real represent the distance covered between the last two readings of the odometer and the location of the car relative to the tape, respectively. Variables c_engine and c_steer model the current lengths of the servo pulses. If both equal zero, the car is stationary with its wheels straight. In a class definition, the number of instances is indicated by placing the number in parentheses after a class name. The classes are shown below:

```
class Data (1) is
    r_dist: real := 0.0; r_tape: real := 0.0;
end Data;

class Output (1) is
    c_engine: real := 0.0; c_steer: real := 0.0;
end Output;
```

Action Clear clears all the variables given in this layer. This is implemented by a parallel assignment in its *body.* Action Read, which has a *participant* of class Data, models the reading of the odometer and the A/D converter. The actual new readings are modeled by two *parameters* r_x and r_y, which have nondeterministic values and do not refer to any objects. In action Control, parameters c_x and c_y are used to model the new values given by the control algorithm. They are assigned to the variables c_engine and c_steer, respectively. The actions are given below:

```
action Clear (D: Data; O: Output) is
when true do
    D.r_dist := 0.0 || D.r_tape := 0.0 ||
    O.c_engine := 0.0 || O.c_steer := 0.0;
end Clear;

action Read (r_x, r_y: real; D: Data) is
when true do
    D.r_dist := r_x || D.r_tape := r_y;
end Read;

action Control (c_x, c_y: real; O: Output) is
when true do
    O.c_engine := c_x || O.c_steer := c_y:
end Control;
```

Because the *guards* of all three actions are identically true, the actions are continually *enabled.* The behavior of the system consists of clearing the variables, reading the inputs and writing the outputs. The order in which these actions are executed is nondeterministic.

3.2.2 Drive States

Layer Drive_States introduces the start/stop switch and specifies the order in which the actions are executed. Class Data is *extended* to hold a state machine d_state, which indicates the actions that are allowed to be executed. The state machine has states start, read and control, the first of which is the default state.

The switch is modeled by variable switch, which has two states, on and off. The state of the switch is changed in action Toggle. When the switch is on in state start, action Start is enabled. It changes the state to read.

The extensions of class Data and the new actions are shown below:

```
extend Data by
    d_state: (start, read, control);
    switch: (off, on);
end Data;

action Toggle (D: Data) is
when true do
    if D.switch'off then
        D.switch → on();
    else
        D.switch → off();
    end if;
end Toggle;

action Start (D: Data) is
when D.switch'on and D.d_state'start do
    D.d_state → read();
end Start;
```

The car is fully operational when the switch is on in states read and control. Likewise, actions Read and Control are refined so that they are enabled correspondingly. Furthermore, by addition of state transition statements D.d_state → control() and D.d_state → read() to Read and Control, respectively, they are executed by turns. The *refined* action Read is given below, where ellipses denote the guard and the body of the original action:

```
refined Read (r_x, r_y : real; D: Data) is
    when ... D.switch'on and D.d_state'read do
    ...
    D.d_state → control();
end Read;
```

Furthermore, action Clear is refined to change the state back to start when the switch is turned off. In this case it implicitly stops the engine and straightens the wheels by clearing all the variables.

3.2.3 *Control Algorithms*

Layer Control_Algorithms treats the controlling of the movement and the steering. The layer defines ten constants, extends class Data, introduces two functions and refines all three actions given in Basic_Actions. The constants and the variables are added due to the control algorithms. Variable e_state represents the state of the engine. The three states power_up, moves and normal are needed since, because of friction, it is necessary to power up until movement is sensed and after that power down slightly to prevent slipping.

The P and PID algorithms are used to compute new values for the engine and steering, respectively. Function PID is shown below (s_P, s_I and s_D are constants):

```
function PID(D: Data) : real is
    return –s_P*D.r_tape + s_I*D.r_tape_ma
        + s_D*(D.r_tape_old – D.r_tape);
end PID;
```

Action Read is refined to include the statements needed by the control algorithm (comments begin with double hyphens):

```
refined Read (r_x, r_y: real; D: Data) is
when ... do          - - the guard is unchanged
    ...
    if (r_x = 0.0) and (D.r_dist = 0.0) then
        D.e_state → power_up();     - - not moving yet
    elsif (r_x > 0.0) and (D.r_dist = 0.0) then
        D.e_state → moves();
    else
        D.e_state → normal();
    end if ||

    - - moving average:
    D.r_tape_ma := ((n – 1)*D.r_tape_ma – D.r_tape)/n ||
    D.r_tape_old := D.r_tape;
end Read;
```

In the guard of action Control, parameters c_x and c_y are bound to the return values of functions P and PID, respectively. If the car has lost the track, it should stop. This is the situation if the absolute value of r_tape is greater than limit, which is treated as a special case in the guard of action Control. The refined action Control is shown below:

```
refined Control (c_x, c_y: real; O: Output; D: Data)
of Control(c_x,c_y,O) is
when ... (c_x = if((abs D.r_tape) > limit)
            then 0.0 else P(D,O) end if)
            and c_y = PID(D) do
    ...
end Control;
```

Furthermore, action Clear is refined to clear variables introduced in this layer.

As already mentioned, the vertical units emerge incrementally while specifying the horizontal layers. In the composed specification, class Data, for instance, consist of all variables added in different layers:

```
class Data (1) is
    - - Basic_Actions:
    r_dist: real := 0.0;
    r_tape: real := 0.0;
    - - Drive_States:
    d_state: (start, read, control);
    switch: (off, on);
    - - Control_Algorithms:
    e_state: (power_up, moves, normal);
    d.r_tape_ma: real := 0.0;
    d.r_tape_old: real := 0.0;
end Data;
```

Obviously, it would have been difficult to come up with such variables directly.

The horizontal architecture of the specification is illustrated in Figure 5. Layer Basic_Actions is common to both concerns, and layers Drive_States and Control_Algorithms treat the functional and control concerns, respectively. The latter layers can be applied in any order on top the former when composing the specification as long as the dependencies are respected, i.e. either Control_Algorithms on top of Drive_States or the other way around.

3.3 Specifying with DisCo

As used in DisCo, superposition is strictly additive, i.e. nothing is removed from the refined specification. This means that all assignments to a variable must be given in the layer introducing the variable. With this restriction, safety properties can be preserved by construction.

Concerning support for specifying non-trivial systems, the DisCo toolset [2] provides animated simulation of DisCo

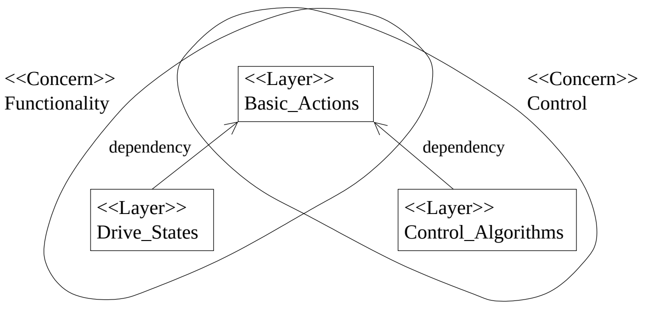

Figure 5: Architecture of the mobile robot specification.

specifications for validation purposes at all levels of refinement. Moreover, assembly of horizontal architectures can be supported by reusable layers as described in [15]. These layers can be seen as behavioral templates that should be formally verified [14].

Related to common software engineering practices, the advantages of better alignment between requirements and specifications are emphasized in the maintenance phase [3]. It should be also noted, that the ideas underlying the approach are insensitive to the notation used, thus offering a foundation for aspect-oriented specification languages. In [12] this was shown using UML.

Model-Driven Architecture (MDA) [8, 7] by Object Management Group (OMG) also includes similar elements to the DisCo approach. In MDA, however, only three pre-defined levels are used, including *computation independent*, *platform independent*, and *platform specific* models. By allowing individual levels of abstraction of MDA to consist of several DisCo layers, one can create a development approach that fits in the guidelines of MDA without compromising rigorousness [20].

4. DISCUSSION

We have shown how two different dimensions of software architecture, which we call vertical and horizontal, can be used for constructing reactive systems. These dimensions are in some sense dual, and they are also incompatible with each other in the sense that it does not seem useful to combine them in a single system of modules. Therefore, the horizontal dimension, which is currently visible in design patterns and aspect-oriented programming, for instance, remains as a crosscutting dimension with respect to conventional vertical units of implementation.

The two dimensions can be combined in different ways. As already discussed, the DisCo approach uses the horizontal dimension as the primary dimension, and provides an incremental specification method for composing specifications. A vertical implementation architecture is, however, anticipated in terms of a high-level view of objects and their cooperation. In contrast, most aspect-oriented approaches, in particular those influenced by AspectJ [22], have taken a more pragmatic approach. There, the primary architectural dimension is based on conventional vertical modularity, and crosscutting aspects are added to it by an auxiliary mechanism for horizontal modularity [6]. Other approaches include problem frames, where the horizontal dimension is used for decomposing a problem into subproblems whose sizes are more manageable [9].

Since layers provide abstractions of the total system, their explicit use seems natural in a structured approach to specification, and also in incremental design of systems. At the programming language level it is, however, difficult to develop general-purpose support for horizontal architectures. This means that a well-designed horizontal structure may be lost in an implementation, or entangled in a basically vertical architecture. However, newer implementation techniques, including aspect-oriented ones in particular, have enabled a wider range of options [1, 21].

We conclude that much work is still needed for making horizontal architectures easier to use in practice. Besides issues concerning implementation, commercial tools for utilizing the horizontal dimension in software design are not available. Thus, it remains a topic of future study to build a tool set where a commonly used notation, such as UML, is used to support the approach in a rigorous way. Accomplishing this may require the introduction of new concepts and terminology, such as those proposed by OMG's Model-Driven Architecture initiative.

5. ACKNOWLEDGMENTS

The authors would like to thank the other members of the DisCo team and Shmuel Katz for the discussions on the subject. This research has been partially funded by Academy of Finland, project ABESIS grant number 100005.

6. REFERENCES

[1] Timo Aaltonen, Joni Helin, Mika Katara, Pertti Kellomäki, and Tommi Mikkonen. Coordinating objects and aspects. *Electronic Notes in Theoretical Computer Science*, 68(3), March 2003.

[2] Timo Aaltonen, Mika Katara, and Risto Pitkänen. DisCo toolset – the new generation. *Journal of Universal Computer Science*, 7(1):3–18, 2001. `http://www.jucs.org`.

[3] Timo Aaltonen and Tommi Mikkonen. Managing software evolution with a formalized abstraction hierarchy. In *Proc. Eight IEEE International Conference on Engineering of Complex Computer Systems*, Greenbelt, MD, USA, December 2002. IEEE Computer Society Press.

[4] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[5] Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[6] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10):29–32, October 2001.

[7] Object Management Group. MDA guide version 1.0.1. OMG Document Number omg/2003-06-01, June 2003. At URL `http://www.omg.org/mda/specs.htm`.

[8] Object Management Group. MDA homepage. At URL `http://www.omg.org/mda/`.

[9] Michael Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems.* Addison–Wesley, 2001.

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

[11] Mika Katara. Composing DisCo specifications using generic real-time events – a mobile robot case study. In Jaan Penjam, editor, *Software Technology, Proc. Fenno–Ugric Symposium*, pages 75–86, Sagadi, Estonia, August 1999. Institute of Cybernetics at Tallinn Technical University (Technical Report CS 104/99). At URL `http://disco.cs.tut.fi`.

[12] Mika Katara and Shmuel Katz. Architectural views of aspects. In *Proc. 2nd International Conference on Aspect-Oriented Software Development*, pages 1–10, Boston, MA, USA, March 2003. ACM Press.

[13] Shmuel Katz and Joseph Gil. Aspects and superimpositions. Position paper in ECOOP 1999 workshop on Aspect-Oriented Programming, Lisbon, Portugal, June 1999.

[14] 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.

[15] Pertti Kellomäki and Tommi Mikkonen. Design templates for collective behavior. In Elisa Bertino, editor, *Proc. ECOOP 2000, 14th European Conference on Object-Oriented Programming*, number 1850 in Lecture Notes in Computer Science, pages 277–295. Springer-Verlag, 2000.

[16] R. Kurki-Suonio. Modular modeling of temporal behaviors. In S. Ohsuga, H. Kangassalo, H. Jaakkola, K. Hori, and N. Yonezaki, editors, *Information Modelling and Knowledge Bases III*, pages 283–300. IOS Press, 1992.

[17] Reino Kurki-Suonio and Tommi Mikkonen. Abstractions of distributed cooperation, their refinement and implementation. In Bernd Krämer, Naoshi Uchihira, Peter Croll, and Stefano Russ, editors, *Proc. International Symposium on Software Engineering for Parallel and Distributed Systems, PDSE'98*, pages 94–102. IEEE Computer Society, 1998.

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

[19] David Lorge Parnas, Paul C. Clements, and David M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–266, March 1985.

[20] Risto Pitkänen. Formal model driven approach to developing enterprise systems. Technical report, Institute of Software Systems, Tampere University of Technology, 2004. At URL `http://disco.cs.tut.fi/reports/formalmdd.pdf`.

[21] Marcelo Sihman and Shmuel Katz. Superimpositions and aspect-oriented programming. *The Computer Journal*, 46(5):529–541, September 2003.

[22] AspectJ home page at `http://aspectj.org`.

[23] DisCo home page at `http://disco.cs.tut.fi`.