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

Mika Katara, Reino Kurki-Suonio and Tommi Mikkonen
Institute of Software Systems
Tampere University of Technology
Finland

Outline

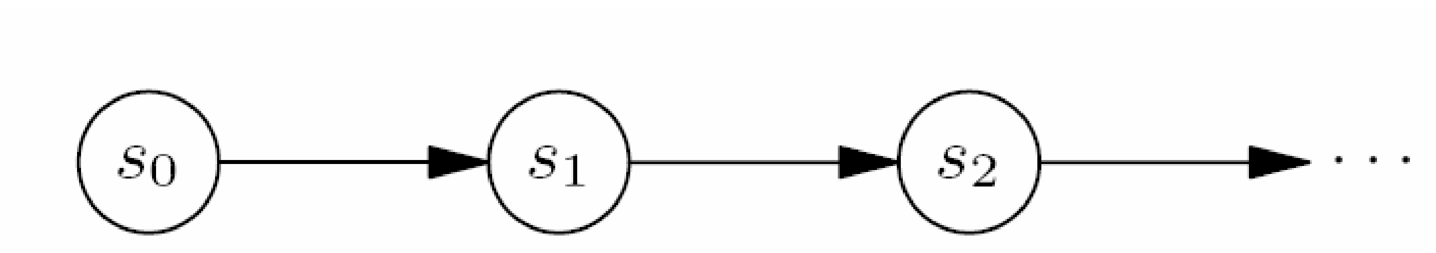
1. Motivation
2. Two dimensions of software architecture
   – vertical units
   – horizontal units
3. Experiences with the DisCo method
4. Conclusions

Motivation

- In order to provide better alignment between conceptual requirements and aspect-oriented implementations, formal specification methods should enable the encapsulation of *logical abstractions*
- *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*

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:



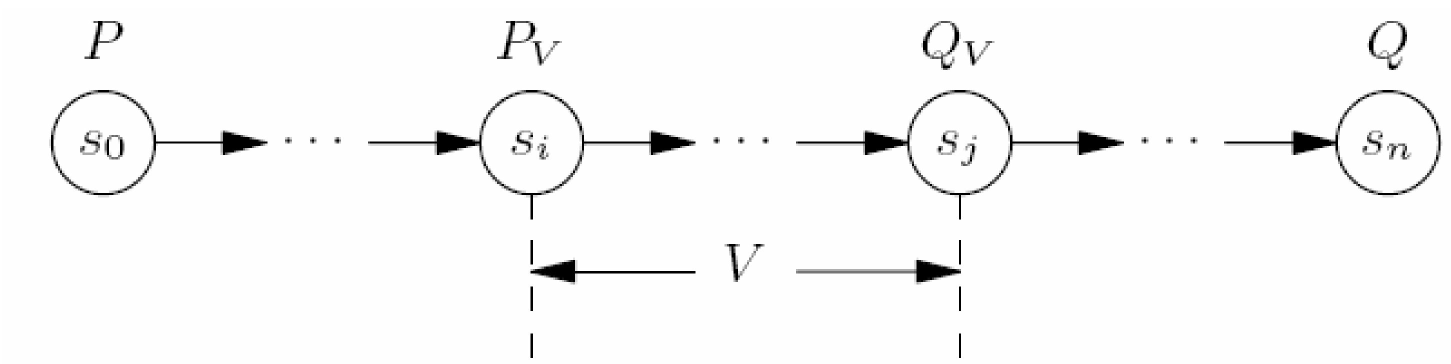
Two dimensions…

- 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 usually nonterminating

Vertical units

- The algorithmic meaning of software, as formalized by Dijkstra, has the desirable property that it can be composed from the meanings of the components in an 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

Vertical…



- 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)*

Vertical…

- More generally, an architecture that consists of vertical units imposes a nested structure of such vertical slices on each state sequence
- In the generation of these sequences, there are two basic operations between architectural units
  - Sequential composition which concatenates state sequences generated by component units
  - Invocation which 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

Vertical…

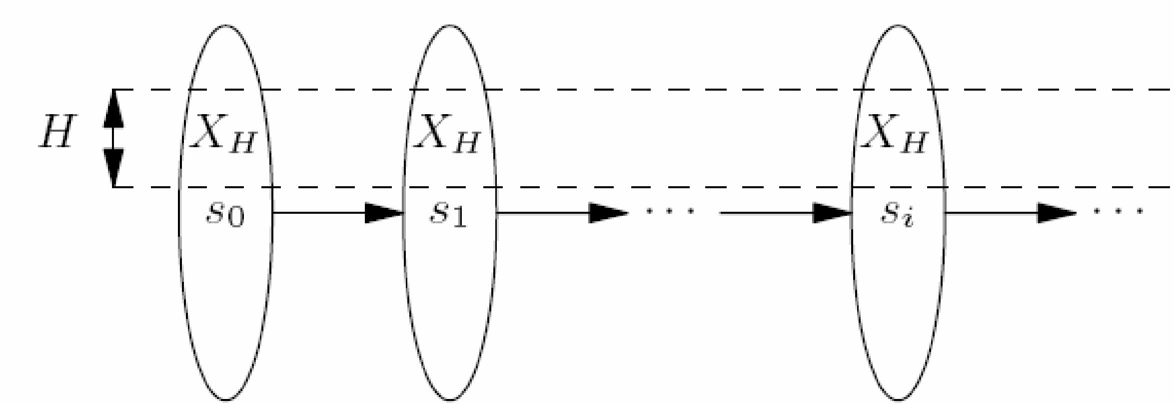
- In current software engineering approaches, this view has been adopted as the basis for designing behaviors of object-oriented systems, leading the focus to *interface operations* that are to be invoked, and to the associated local precondition-postcondition pairs

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*

Horizontal…

- 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:



Horizontal…

- 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

Experiences with the DisCo method

- 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

Experiences…

- 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 abstractions of the system, they serve in capturing concepts of the problem domain

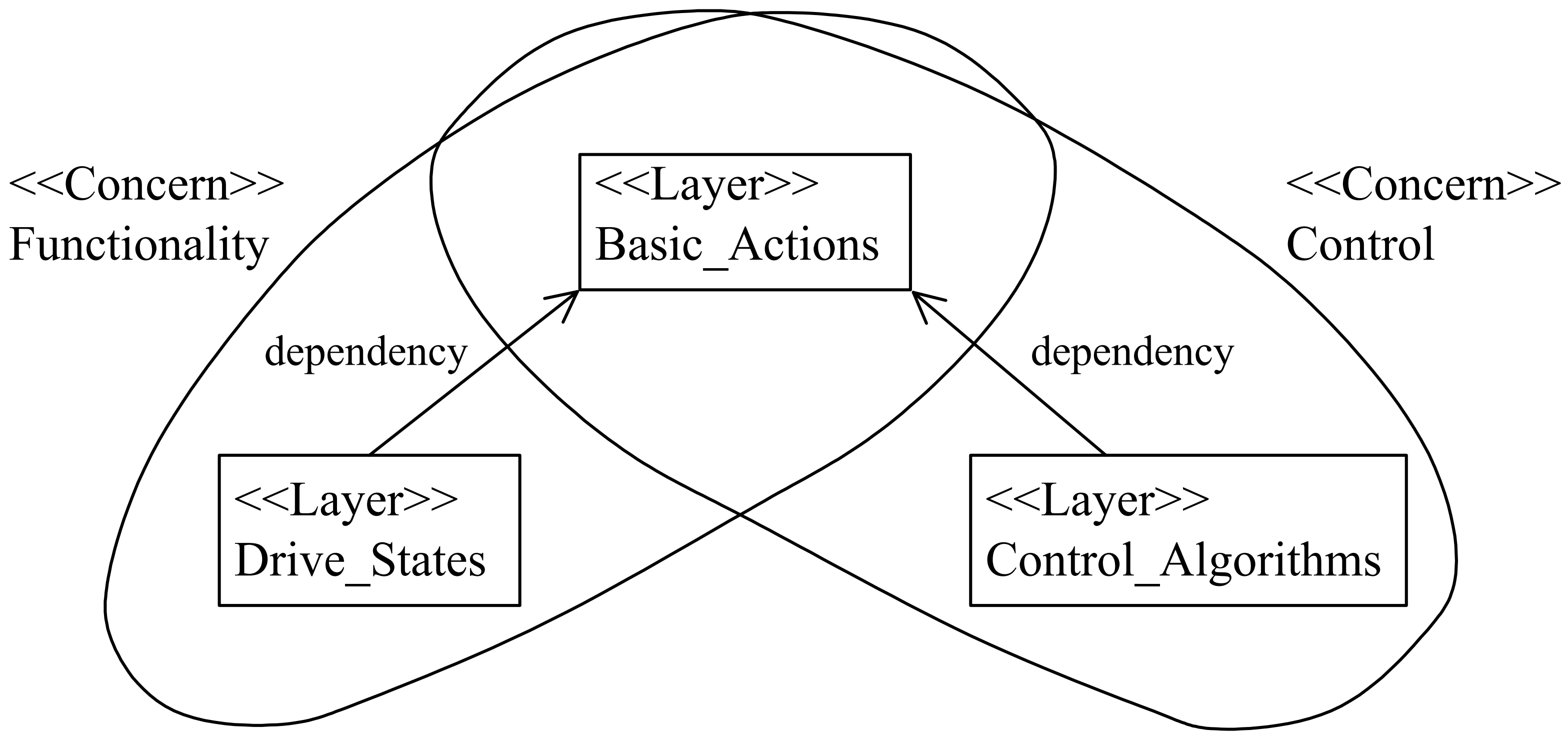
Example: mobile robot

- Mobile robot is a small microcontroller-based car
- 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 sensors, and from an odometer
  - The outputs are PWM (Pulse Width Modulation) signals that drive the two servo motors controlling the steering and the movement
- There is also a switch, which is used to start and stop the car

Example…

- There are two main *concerns* that need to be addressed:
  - Basic functionality of the car including starting and stopping
  - 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

Example…



<<Concern>>
Functionality
<<Layer>>
Basic_Actions
<<Concern>>
Control
dependency
dependency
<<Layer>>
Drive_States
<<Layer>>
Control_Algorithms

Example...

```
layer ba is

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 (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;

end ba;
```

```
layer ca is import ba;

extend Data by
  r_tape_ma: real;
  r_tape_old: real;
  e_state: (power_up, moves, normal);
end;

refined Read (r_x, r_y: real; D: Data) is
when ... do
  ...
  if (r_x = 0.0) and (D.r_dist = 0.0) then
    D.e_state -> power_up();
  elsif (r_x > 0.0) and (D.r_dist = 0.0) then
    D.e_state -> moves();
  else
    D.e_state -> normal();
  end if ||
  D.r_tape_ma := ((8.0 - 1.0)*D.r_tape_ma -
    D.r_tape)/8.0 ||
  D.r_tape_old := D.r_tape;
end Read;

end ca;
```

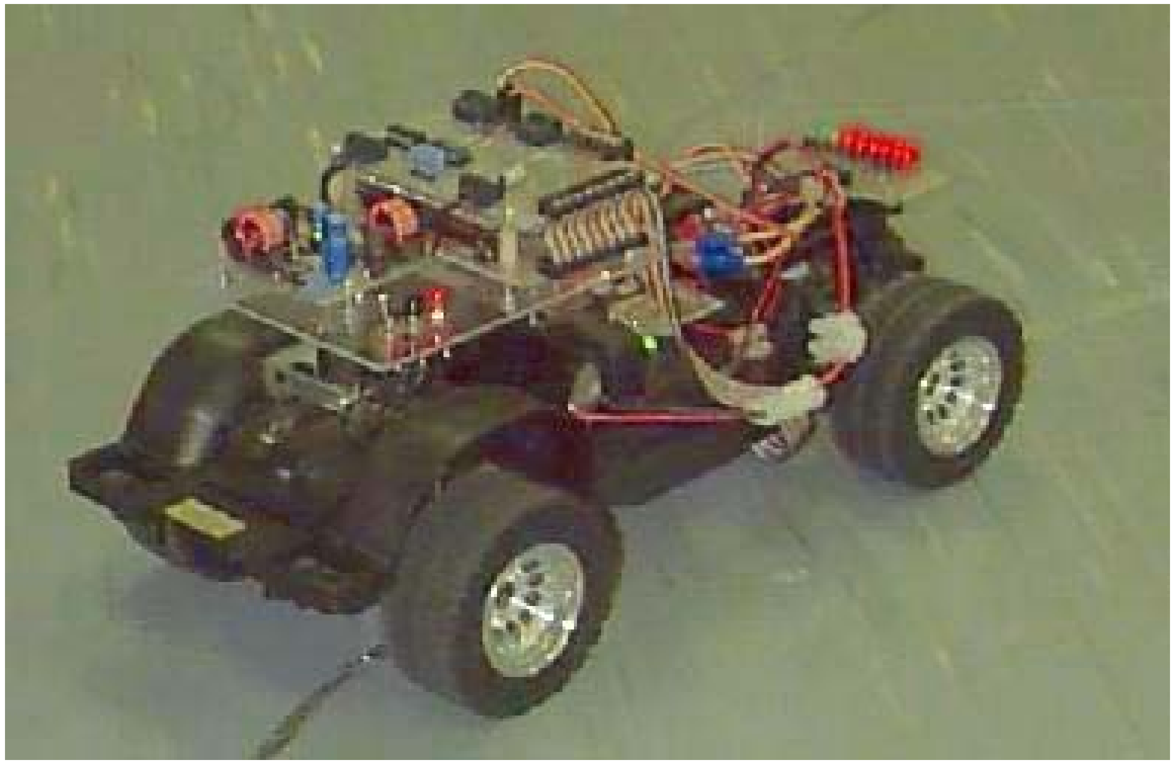
Conclusions

- The two dimensions of architecture are in some sense dual to each other
  - 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*

Conclusions…

- 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

Questions?



Further information & tools available at disco.cs.tut.fi