Separation of Context Concerns — Applying Aspect Orientation to VDM

Naoyasu Ubayashi[1] and Shin Nakajima[2]

[1] Kyushu Institute of Technology, Japan
[2] National Institute of Informatics, Japan
`ubayashi@acm.org, nkjm@nii.ac.jp`

Abstract. Separation of concerns is important to reduce the complexity of software design. This paper examines a software development method starting with the feature-oriented modeling method to have VDM-based formal design. In order to overcome the problem that a feature may be scattered over the VDM design description, the notion of the aspect is adapted to propose AspectVDM. The identified features are concisely represented in AspectVDM to demonstrate modular descriptions of cross-cutting concerns in VDM.

1 Introduction

Separation of concerns is recognized important to reduce the complexity of software design [10]. It is also true the clear hierarchical decomposition is not possible because some concerns may cross-cut others. The idea of aspect has been introduced to remedy this problem of cross-cutting concerns in a systematic way [1][6].

The concerns are related to a wide variety of viewpoints, some of which are strongly related to the requirements of the system. Identifying the features having much impact on the system design is not trivial. In particular, for the case of embedded systems such as home electrical appliances or control equipments, one originating from the *working environment* is dominant [4], but easy to be overlooked. These systems should show behavior reacting to a certain change in their environment conditions or properties of the context. The design of the embedded system should take into account of all possible such changes, which sometimes results in a set of description fragments spreading over a lot of modules. A systematic approach to both identifying such context properties and having a clear rigorous design description is called for.

This paper proposes to employ the feature-oriented modeling method [5] and the VDM-based formal design [2][3] with the notion of the aspect [1][6][9]. The overall method provides a systematic way to find the concerns relating to the properties of the context and further to have aspect-oriented VDM descriptions, AspectVDM descriptions. The identified context concerns have clear correspondence with the aspect module in AspectVDM.

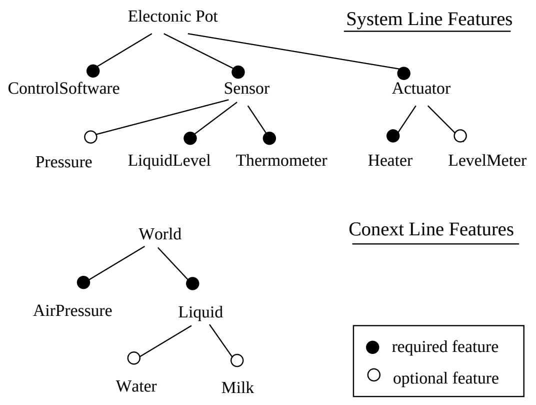

Fig. 1. Feature Tree for Electric Pot and World

2 Separation of context concerns

This section illustrates the notion of *separation of context concerns* and its importance in the system requirement stage. The discussion uses a concrete example of an electric pot — in this case, its context is water.

2.1 Context-aware Feature-Oriented Modeling

An electric pot controller is an embedded software for boiling water. For simplicity, the discussion here looks at the following requirements: 1) the pot has three hardware components including heater, thermistor, and water level sensor; 2) the pot controls the water temperature by turning on or off the heater; 3) the pot changes its mode to the heat-retaining mode when the temperature becomes 100 Celsius; and 4) the pot observes the water volume from the level sensor that detects whether the liquid is below or above the specified level.

In order to make the early stage of the development systematically, Feature-Oriented Modeling (FOM) [5] has been proposed especially for those such as embedded systems, which is aimed to supprt Software Product Line (SPL). Since, for example, a consumer appliance has a wide variety of similar but different products, its development requires to identify a set of commonalities and variabilities. Ideally, a product is developed by assembling all the commonalities and some subset of variabilities for fulfilling the requirements of the very product. FOM is one of the modeling method used in SPL.

FOM provides a tree notation to explicitly represent the relationships among the identified features; some are commonalities and others variabilities. The features are arranged in a tree where a lower node becomes a constituent of the upper one. Figure 1 is a portion of the feature tree for the electric pot. For example, the top node, a `Electric Pot` feature is decomposed into `ControlSoftware`, `Sensor`, and `Actuator`. `Actuator` is, in turn, expanded into a mandatory `Heater` and an optional `LevelMeter`.

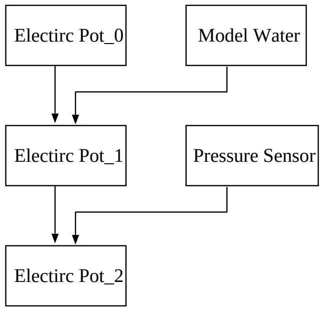

Fig. 2. Development Scenario

The method unique to this paper is to have *Context Line Feature* explicitly separated from the usual *System Line Feature* mentioned above. Figure 1 shows that `World`, which may provide the *working environment* for the pot, has various features such as `AirPressure` or `Liquid`. Apparently, these features are not unique to the pot, but can be defined generally in terms of the well-known laws of Physics. At the same time, *Context Line Feature* should be taken into account in the design of the electric pot product.

2.2 Design in VDM

Since *System Line Feature* and *Context Line Feature* are based on quite different viewpoints, the portions of the design reflecting each one are expected to be clearly separated. They are actually interwoven because the control software should consider the properties of the physical water.

The proposed development process is incremental and thus simple since a small new addition is made at a time. As shown in Figure 2, it starts with a small core electric pot (`Electric Pot_0`), and adds it the properties of water (`Model Water`) to have the second version (`Electric Pot_1`). It is followed by adding `Pressure Sensor`, which is not discussed here.

Step 1: model system specifications
The base description of the electric pot (`Electric Pot_0`) would be obtained first by identifying the state definition and the operations to change the state.

Figure 3 gives a rough sketch of the state and how it is changed due to the operations. The state is composed of temperature ranging from 0 to 100 (`<Hot>` is meant to represent a certain *hot* water), water level (below or above the base volume), and power switch (on or off). There are four operations including `PourIn` (pour the water), `PourOut` (pour out the water), `SwitchOn` (turn on a switch), `SwitchOff` (turn off a switch), and `Boil` (heat up the water). `Boil` invokes function `incTem` (raise the temperature) whose definition is omitted.

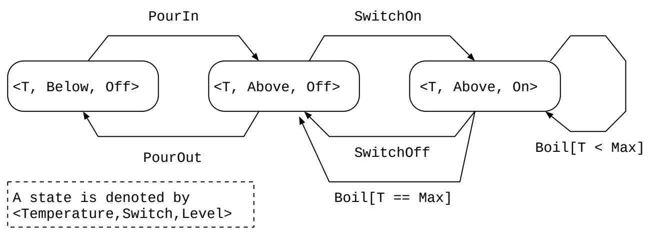

Fig. 3. State transitions of system(pot)

```
[List 1]
types
  Tem = <Zero> | <Room> | <Hot> | <Max> ;
  Level = <Below> | <Above> ;
  Switch = <On> | <Off> ;
state Pot of
   temp : Tem
   liquid : Level
   heat : Switch
inv  pot == (pot.liquid = <Below>) => (pot.heat = <Off>)
init pot == pot = mk_Pot(<Room>,<Below>,<Off>)
end
operations
  PourIn()
    ext wr liquid : Level
        rd heat : Switch
    pre  (liquid = <Below>) and (heat = <Off>)
    post (liquid = <Above>) ;
  PourOut()
    ext wr liquid : Level
        rd heat : Switch
    pre  (liquid = <Above>) and (heat = <Off>)
    post (liquid = <Below>) ;
  Boil()
    ext wr temp : Tem
        rd liquid : Level
        wr heat : Switch
    pre  (liquid = <Above>) and (heat = <On>)
    post (   (temp~ = <Max>) => (heat = <Off>))
     and (not(temp~ = <Max>) => (temp = incTem(temp)))
  SwitchOn()
    ext wr heat : Switch
        rd liquid : Level
    pre  (liquid = <Above>) and (heat = <Off>)
    post (heat = <On>) ;
  SwitchOff()
    ext wr heat : Switch
        rd liquid : Level
    pre  (liquid = <Above>) and (heat = <On>)
    post (heat = <Off>) ;
```

Step 2: model context specifications

The Context Line Feature is transformed into the `Model Water` in Figure 2. It describes the physical properties in regard to heating up or cooling down.

The VDM description of `Model Water` is shown below to use `type` and `function`. The status of the water is composed of temperature, water volume, and air pressure. Here, only 1.0 and 0.53 atmosphere are considered as an example. Two of the functions are important; `heatUp` (heat up the water), `critical` (get the value of a critical boiling point). Other auxiliary functions are omitted.

```
[List 2]
types
  Vol = <Empty> | <Little> | <Large> | <Full> ;
  Tem = <Zero> | <Room> | <Hot> | <Max> ;
  Water :: t : Tem
           v : Vol
           p : real
  inv mk_Water(x,y,z) == (x in set { <Zero>, <Room>, <Hot>, <Max> })
                     and (y in set { <Empty>, <Little>, <Large>, <Full> })
                     and (z in set { 1.0, 0.53 })
functions
  heatUp (w : Water) r : Water
    pre  w.v <> <Empty>
    post  (ltTem(w.t, critical(w.p))
           => (r = mk_Water(incTem(w.t), w.v, w.p)))
     and  ((w.t = critical(w.p))
           => (r = mk_Water(w.t, decVol(w.v), w.p))) ;
  critical(p : real) r : Tem
    post ((p = 1.0) => (r = <Max>)) and ((p = 0.53) => (r = <Hot>)) ;
```

Step 3: compose the system and context specifications

The second version of the electric pot (`Electric Pot_1`) in Figure 2 is obtained by combining `Electric Pot_0` and `Model Water`. The composed description is considered to describe the actual situation when the electric pot is used. In other word, it simulates the situation where the electric pot is filled with the *physical* water. Therefore, `Electric Pot_1` can be considered as a faithful design to take into account the working environment. Below shows a fragment of `Electric Pot_1`, which is obtained by human editing. What differs from `Electric Pot_0` are marked with `*`.

```
[List 3]
  state Pot of
     temp : Tem
     liquid : Level
     heat : Switch
*    water : Water
   inv  pot == (pot.liquid = <Below>) => (pot.heat = <Off>)
*          and (pot.temp = pot.water.t)
*          and ((pot.liquid = <Below>) <=> (ltVol(pot.water.v, <Little>)))
   init pot ==
*       pot = mk_Pot(<Room>,<Below>,<Off>,mk_Water(<Room>,<Little>,1.0))
*    or pot = mk_Pot(<Room>,<Below>,<Off>,mk_Water(<Room>,<Little>,0.53))
   end
  operations
   PourIn()
     ext wr liquid : Level
         rd heat : Switch
*        wr water : Water
     pre  (liquid = <Below>) and (heat = <Off>)
*    post (liquid = <Above>) and (water.v = <Large>);
   PourOut()
     ext wr liquid : Level
         rd heat : Switch
*        wr water : Water
     pre  (liquid = <Above>) and (heat = <Off>)
```

```
*    post (liquid = <Below>) and (water.v = <Little>);
   Boil()
     ext wr temp : Tem
         rd liquid : Level
         wr heat : Switch
*        wr water : Water
     pre  (liquid = <Above>) and (heat = <On>)
     post (   (temp~ = <Max>) => (heat = <Off>))
      and (not(temp~ = <Max>) => ((temp = incTem(temp~))
*     and (water = heatUp(water~))))
```

There are two kinds of modifications: 1) adding a reference to the context (i.e. water), and modifing the associated *mk* patterns; 2) adding logical expressions that represent the properties of the context to invariants and pre/post-conditions. For example, new predicates are *added* (logical $\wedge$) to the invariant of the state `Pot`. They represent that what the pot looks at is the faithful representation of the real water contained in it.

3 Introducing Aspect into VDM-SL

3.1 Aspect-Oriented Development

As presented in the previous section, writing down VDM descriptions to follow the idea of *separation of context concerns* requires to edit various parts of the base description (`Eclectic Pot_0`). In a word, the modification is scattered. The process is not systematic as well as error-prone. By introducing the notion of the aspect in an explicit manner, the process is expected to be systematic, and possibly automatic.

This section proposes AspectVDM that follows the idea of Join Point Model (JPM) in AspectJ, a representative of AOP languages [6][7]. Its aspectual notion is captured by the use of *advice* and *pointcuts* to modify behavior that crosscuts the structure of the base program.

In AspectVDM, *join points* are a collection of elements in the description at which the advice may be woven. Such join points are specified by a *pointcut*. An *advice* is a change in the description, performed at the join points in the pointcut. All the *pointcut* and *advice* for a particular concern are contained in *Aspect* module. Figure 4 presents a schematic illustration of the relationship between *Aspect* and the base VDM. Below illustrates a fragment of *Aspect* module for the electric pot, of which advices work on the `Electric Pot_0` to produce the `Electric Pot_1` equivalent.

As discussed in the preceeding section, the state and operations in the base would be modified. With this in mind, AspectVDM is so chosen to provide the pointcut designators working on preconditions, postconditions, invariants, and initialization. The aspect module also supports *open class*, a mechanism for adding elements to the definition of a record type or a state [6].

```
[List 4]
aspect pot_water of
  Pot.water : Water
  pointcut potinv() : invariant(Pot.pot)
```

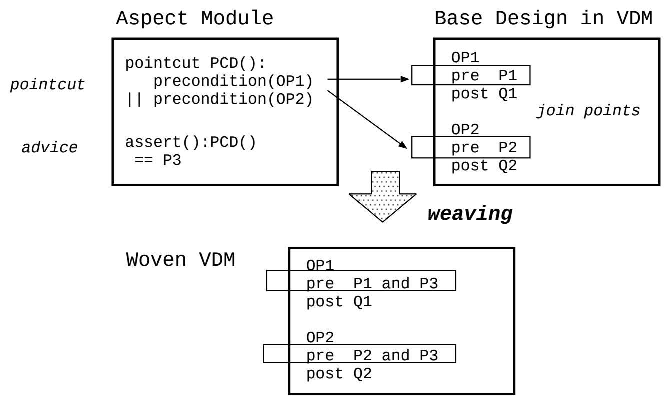

Fig. 4. AspectVDM JPM

```
  pointcut potinit() : init(Pot.pot)
  assert() : potinv() == (pot.temp = pot.water.t)
     and ((pot.liquid = <Below>) <=>(ltVol(pot.water.v, <Little>)))
  replace() : potinit() ==
        pot = mk_Pot(<Room>,<Below>,<Off>,mk_Water(<Room>,<Little>,1.0))
     or pot = mk_Pot(<Room>,<Below>,<Off>,mk_Water(<Room>,<Little>,0.53))
end
```

An aspect definition begins with the keyword `aspect` followed by an aspect name, *pot_water* in this case. The Line 2 `Pot.water` is an open class declaration for adding the element `water` to the `Pot` state. The type of `water` is `Water`. A pointcut is declared by the keyword `pointcut` followed by a pointcut name, a colon, and a pointcut body specified by a pointcut designator. There are two pointcut definitions in List 4. The `potinv` is a pointcut for capturing the invariant `pot` specified in the `Pot` state. The `potinit` is a pointcut for capturing the initialization `pot` specified in the `Pot` state.

Advice is defined with the keyword such as `assert()`. The keyword is followed by a colon, a related pointcut name, and an advice body. The advice for the `potinv` pointcut appends a logical expression at Line 6 - 7 to the original invariant. The advice for the `potinit` pointcut replaces the original `Pot.pot` initialization by Line 9 - 10.

Although the above example does not mention at all, an *aspect* module can contain a complete definition of an operation, a type, or a function. It sometimes requires a new function to write down concisely the formula in `pre` or `post` of an existing operator in the base.

Weaving this aspect and the base pot specification (`List 1`), the `List 3` specification is obtained. It demonstrate that the context specification (`List 2`) can be separated from the base description of the pot, and `List 3` is not necessary to wrote down manually.

The above example also shows that the modification on `pre` and `post` seems freely done, which may lead to logical inconsistency. A certain kind of logical checking is mandatory for the aspect to be meaningful. The next section will give some discussion on this point.

Last, a complete set of the pointcut designators and the advices is not fixed at the time of writing this paper; the advice may include `retract` as well as `assert` for an operator's `pre` and `post`. For example, in order to change `p and q` to be `((p => x) and (not p => y)) and q`, for the original base description, first `p` is `retract`ed and then `((p => x) and (not p => y))` is `assert`ed. The definition, both syntax and semantics, of the pointcut designators and the advices is still under investigation.

3.2 Weaving and Proof Obligation

With a slight look, the weaving seems a simple syntactical transformation of the VDM description. Weaving in AspectVDM, however, is not just a syntactical transformation alone. How proof obligation is generated should be considered.

As the effect of the advice, the base description will schematically be changed after the weaving as below.

– For `state`, its component may be added : `S` changes to $\mathtt{S} + \delta\mathtt{S}$.
  - For `init`, the initialization pattern may be completely changed : `K(S)` changes to $\mathtt{L(S} + \delta\mathtt{S)}$.
  - For `inv`, the invariant may be added : `I(V)` changes to $\mathtt{I(V)}\wedge\mathtt{J(V} + \delta\mathtt{V)}$.

– For an operation `Op`, the pre- and post-conditions may be modified :
  - For `pre`, `P` changes to `P'`.
  - For `post`, `Q` changes to `Q'`.

In the above, `V` represents a set of component names defined in `S` and $\mathtt{V} + \delta\mathtt{V}$ is a set of names in $\mathtt{S} + \delta\mathtt{S}$. The original invariant in the base `I` looks at `V` and the added part `J` sees $\mathtt{V} + \delta\mathtt{V}$.

In order for the aspect description to be consistent, the following formula should be satisfied where `S'` refers to $\mathtt{S} + \delta\mathtt{S}$.

– The addition to `inv` is valid : $\mathtt{I(V)}\wedge\mathtt{J(V} + \delta\mathtt{V)}$.
– The modification to `pre` is valid : $\forall\mathtt{S'} \mid \mathtt{P'}$.
– The modification to `post` is valid : $\forall\mathtt{S'} \mid \mathtt{Q'}$.

Further, since an operation `Op` after weaving (denoted by $\mathtt{Op}^w$) should be valid in the context where the original base `Op` is valid, the formula

$$\forall\mathtt{S'} \mid \mathtt{P} \Rightarrow \mathtt{P'}$$

for $\mathtt{Op}^w$ should be satisfied. Note that `P` of `Op` is evaluated under `S'`.

All the operations being not woven are expected to be valid after the weaving. It implies that the proof obligations before the weaving are preserved for such operations. However, an addition to the invariant ($\mathtt{I(V)}\wedge\mathtt{J(V} + \delta\mathtt{V)}$) may invalidate their pre- or post-conditions. They should be worked on again to generate new proof obligations. Therefore, a further rule is required to determine how such affected operations are collected.

Let `ext(Op)` be the collection of variable names appeared in `ext` of `Op` and `v-name(J)` be the collection of variable names appeared in `J`, then all `Op`'s satisfing the formula

$$\texttt{v-name(J)} \cap \texttt{ext(Op)} \neq \phi$$

should be re-analyzed to generate proof obligations again where

$$\texttt{v-name(J)} \subseteq \texttt{J(V} + \delta\texttt{V)} \text{ and } \texttt{ext(Op)} \subseteq \texttt{V}.$$

Further, for such an `Op`, if the formulas

$$\forall\ \texttt{S'} \mid (\texttt{P} \wedge \texttt{I}) \wedge \texttt{J} \text{ and } \forall\ \texttt{S'} \mid (\texttt{Q} \wedge \texttt{I}) \wedge \texttt{J}$$

are not satisfied, then the current aspect definition is not correct in that the added invariant may violate either `P` or `Q` or both of such `Op`. The aspect should be re-written to include modifications on the `P` or `Q`.

Last, although not mentioned on `type` and `function`, what are discussed in regard to `state` and `operation` are equally applicable to them.

4 Discussion and Conclusion

This paper first pointed out the importance of *separation of context concerns*, and discussed that a formal design in VDM was constructed in a systematic manner with the notion of the aspect. AspectVDM, proposed in the current paper, is supposed to be the first such an attempt. Below presents some discussions on the proposed AspectVDM.

First, refinement and weaving are similar in that both concerns with the model transformations. However, in the refinement, the description becomes concrete toward the programming level implementation. On the other hand, the woven description stays at the same level as before because the base and the aspect look at the same abstraction level. Further, weaving provides a basis for the incremental development of the design. The key point of the aspect is that it provides an explicit language construct *Aspect* module which contains all the necessary information to modify the base description. What, otherwise scattered, can be described in one *Aspect* module.

Second, VDM-SL allows both *explicit* and *implicit* specifications, the latter of which are studies in this paper. Since *explicit* specification is meant to execute, the notion of the aspect would be very similar to that found in AspectJ. Studying how the aspect in two specification styles are related is important. In addition, although this paper adapts JPM motivated by AspectJ, other approaches have been discussed in the *aspect* research community [7][11][12]. Studying what notion of aspect would be adequate for VDM-SL is important direction to pursue.

Third, there are some work, not on VDM, but related to AspectVDM. As for integrating the aspect notion with the exsiting formal specification languages, Yu et al propose an aspect extension of Z [14] and Object-Z [15]. Yamada and

Watanabe [13] introduce the notion of the aspect into JML. JML is used to write down the Design-by-Contract style specifications for Java methods essentially consisting of pre- and post-conditions. With an appropriate JPM, pre- and post-conditions of more than one methods are modified at a time, which is also possible in AspectVDM. They, however, discuss nothing on the proof obligation. Masuhara et al [8] integrates the aspectual notion with strongly-typed functional programming language Caml, which looks at the technical problems similar to those that `type` and `function` may have in AspectVDM.

Although AspectVDM is far from a complete language in that the definition, both syntax and semantics, of the pointcut designators and the advices is still under investigation. This paper is expected to call for further discussions on the notion of the aspect in the VDM community.

References

1. Elrad, T., Filman, R.E. and Bader A.: Aspect-oriented programming, *Comm. ACM*, vol.44, no.10, pp.29-32, 2001.
2. Fitzgerald, J. and Larsen, G. P.: *Modeling Systems, Practical Tools and Techniques in Software Development*, Cambridge University Press, 1998.
3. Fitzgerald, J., Larsen, G. P., Mukherjee, P., Plat, N., and Verhoef, M.: *Validated Designs for Object-oriented Systems*, Springer Verlag, 2005.
4. Greenspan, S., Mylopoulos, J., and Borgida, A.: Capturing More World Knowledge in the Requirements Specification, In *Proc. ICSE'82*, pp.225-234, 1982.
5. Kang, K. C., Lee, J., and Donohoe, P.: Feature-Oriented Product Line Engineering, *IEEE Software*, Vol. 9, No. 4, pp.58-65, 2002.
6. Kiczales, G., et al.: Aspect-Oriented Programming, In *Proc. ECOOP'97*, pp.220-242, 1997.
7. Masuhara, H. and Kiczales, G.: Modeling Crosscutting in Aspect-Oriented Mechanisms, In *Proc. ECOOP 2003*, pp.2-28, 2003.
8. Masuhara, H., Tatsuzawa, H., and Yonezawa, A.: Aspectual Caml: an Aspect-Oriented Functional Language, In *Proc. ICFP'05*, pp. 320-330, 2005.
9. Nakajima, S. and Tamai, T. : Lightweight Formal Analysis of Aspect-Oriented Models. Workshop on Aspect-Oriented Modeling at UML2004, 2004.
10. Parnas, D. : On the Criteria To Be Used in Decomposing Systems into Modules. *Comm. ACM*, Vo. 15, No. 12, pp.1053-1058, 1972.
11. Tarr, P., Ossher, H., Harrison, W. and Sutton, S.M., Jr.: N Degrees of Separation: Multi-dimensional Separation of Concerns, In *Proc. ICSE'99*, pp.107-119, 1999.
12. Ubayashi, N., Moriyama, G., Masuhara, H., and Tamai, T.: A Parameterized Interpreter for Modeling Different AOP Mechanisms, In *Proc. ASE 2005*, pp.194-203, 2005.
13. Yamada, K. and Watanabe, T. : An Aspect-Oriented Approach to Modular Specification of Java Component. In *Proc. IASTED SE2005*, 2005.
14. Yu, H., Liu, D., Yang, L., and He, X. : Formal Aspect-Oriented Modeling and Analysis by AspectZ. In *Proc. SEKE'05*, pp.175-180, 2005.
15. Yu, H., Liu, D., Shao, Z., and He, X. : Modeling Complex Software Systems Usning an Aspect Extension of Object-Z. In *Proc. SEKE'06*, pp.11-16, 2006.