# Multi-Aspect System Analysis Using State Machines Extracted from Specifications in VDM–SL

**Kengo Miyoshi, Satoru Hirachi, Shigeru Kusakabe, and Keijiro Araki**
Department Computer Science and Communication Engineering,
Graduate School of Information Science and Electrical Engineering,
Kyushu University
6-10-1 Hakozaki, Higashi-ku, Fukuoka 812-8581, JAPAN
kengo, h_satoru, kusakabe, araki@ale.csce.kyushu-u.ac.jp

**ABSTRACT**

Model-oriented formal specification languages such as VDM–SL is useful to describe functional requirements of the target systems. However, since single-aspect analysis is not enough to make reliable specifications, we also use other approaches such as model checking to analyze dynamic aspects of the system as a part of multi-aspect analysis. In this paper, we discuss our approach to extract state machines from specifications in VDM–SL. We can analyze both static and dynamic aspect by using these two different kinds of specification languages, model-oriented and state-machine languages in an integrated manner.

**Keywords**

Formal Methods, Multi-Aspect Analysis, Formal Specification, VDM–SL, State Machine

## INTRODUCTION

Formal methods play a very important role in system development. There are many kinds of specification languages for formal methods in the literature. One of the most popular types of specification languages is model-oriented specification languages such as Z[1, 2] and VDM–SL(Vienna Development Methods Specification Language)[3, 4].

Authors mainly use VDM–SL to describe the functional requirement of the system. It supports a mathematical model built from simple data types, along with functions and operations on them. One of the important features of VDM–SL is VDM Tools, a rich set of tools[5]. There are not only syntax and type checker but interpreter for specifications in VDM–SL and generator for C++ and UML.

However, using only one kind of formal specification languages is not enough. While VDM-SL is very useful to describe the functional requirements of the system, multi-aspect analysis is important to construct reliable specifications. Thus, although we mainly use VDM-SL in our system development, we use different types of means such as model checking to analyze dynamic aspects of the system as a part of multi-aspect analysis. In this paper, we propose a method to extract state machines written in FSP (Finite State Processes) from specifications in VDM–SL. After trying to generate state machines in PROMELA, a language for a model checking tool SPIN[7], we find FSP is more suitable for our purpose. FSP is a textual notation of finite state machine models for LTSA (Labeled Transition System Analyzer)[8]. We can analyze both static and dynamic aspect by using specifications in these two different kinds of specification languages in an integrated manner.

There are so-called “modern model checking,” in which state machines are extracted from programs in conventional programming languages. However, as programs in conventional programming languages have a number of states, checking state machines extracted by such approaches faces to state space explosion problem. We start from a specification in VDM–SL and analyze the specification through a variant of dataflow analysis. By starting from abstract but rigorous formal specifications, we can construct more reliable state models with less states compared to other approaches starting from conventional programming languages. Then we generate a state machine model in FSP and compose it with other state machine models to construct models of more complex system if necessary. We use the state machine models to check the dynamic behavior of the system using LTSA, a tool for FSP. In this paper, we use an example of traffic signal in a textbook[3]. By using our approach, we can see in a comprehensive way that the original specification has no description of fairness.

## MULTI-ASPECT ANALYSIS

We use not only VDM–SL but also other approaches such as model checking to analyze multi-aspect of the target systems as shown in Figure 1.

Reading specifications in VDM–SL is rather easy once we

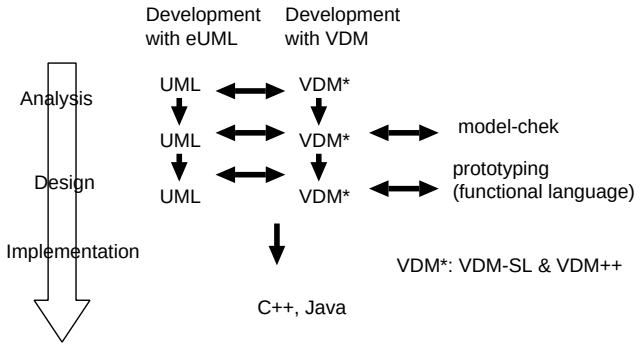


Figure 1: System development using multi-aspect analysis

have become familiar with the language but describing a formal specification in VDM–SL from scratch is still very difficult even if we are familiar with the language. As an introduction step to construct formal models, we construct models in intuitive but informal specification language UML (Unified Modeling Language)[10]. We also use functional programming languages to prototype formal specifications. Since functional programming languages are declarative one, it is rather easy to make prototypes in functional programming languages from specifications in VDM–SL.

Although we can make a state-base model in VDM–SL by using operations, analyzing dynamic aspects of the system using different approach such as model checking is also useful to make reliable specifications of the target system. VDM–SL is not so good at describing and checking dynamic behaviors of the system. Different from Z, VDM–SL has an interpreter in VDM–Tools which support interpretation of specifications in VDM–SL in addition to syntax and type checks. However, all specifications are not supported by the interpreter in VDM–Tools. We can use state-based formal models effectively to describe and check dynamic aspects of the system. State models are also very popular to analyze the system requirements, but they are not so good at describing functional requirement. We intend to use both model-oriented specifications and state machine specifications to analyze multi-aspects of the target system in an integrated fashion.

**FSP and LTSA**

State machines are useful to model behaviors of the target system. In the literature, several kinds of state machines have been studied. Purely state-based formalism such as Kripke structures[6] are often used to model and specify systems. However, we use a kind of labeled transition systems (LTS) in which state transitions are labeled by actions. We are interested in a cooperative modeling method using both model-oriented formal specification languages and state machine specification languages. In the specification in model-oriented language, state changes are performed implicitly by functions or explicitly by operations. We use labeled transition system, as we want to recognize the correspondence between a function (or an operation) and an action in the labeled transition systems. Specifically, we use FSP notation and LISA Tool to analyze the state model of the system.

As we show later, we tried to use PROMELA but we find FSP is more suitable for our purpose. PROMELA is a language for a model checking tool SPIN. Its description is rather low level like programming in C, and we have to consider message passing and corresponding message buffers. After going through type check of VDM–SL specification, we have to program different types in PROMELA. Since we want to examine static and dynamic aspect of the target system at the same abstraction level, the gap of description level between VDM–SL and PROMELA is not preferable. LTSA is a verification tool for concurrent systems. It checks whether the specification of a concurrent system satisfies the properties required of its behavior. In addition, LTSA supports specification animation through GUI to facilitate interactive exploration of system behavior. By using FSP and LTSA, we can focus on dynamic aspect of the target system at an abstract level.

A system in LTSA is modeled as a set of interacting finite state machines, and the properties required of the system are also modeled as state machines. LTSA performs compositional reachability analysis to exhaustively search for violations of the desired properties. More formally, each component of a specification is described as an LTS, which contains all the states a component may reach and all the transitions it may perform. However, explicit description of an LTS in terms of its states, set of action labels and transition relation is cumbersome for other than small systems. Thus, LTSA supports a process algebra notation (FSP) for concise description of component behavior. The tool allows the LTS corresponding to an FSP specification to be viewed graphically.

**EXTRACTION OF STATE MACHINE**

Our method extract state machines from specifications in VDM-SL by focusing on definition and modification of variables in the specification.

First we use a small example of a traffic signal whose specification is shown below.

```
types

  Light = <Red> | <Amber> | <Green>;

functions

  ToGreen: Light -> Light
  ToGreen(light) == <Green>
  pre light = <Red>;

  ToAmber: Light -> Light
  ToAmber(light) == <Amber>
  pre light = <Green>;

  ToRed: Light -> Light
  ToRed(light) == <Red>
  pre light = <Amber>;
```

The light of the traffic signal is associated with the type `Light` whose value is `<Red>`, `<Amber>`, or `<Green>`. The change to `<Green>` is only possible from `<Red>`. Thus the function `<ToGreen>` has a precondition claiming `light = <Red>`. Similar preconditions is claimed for the function `ToAmber` and `ToRed`.

**Model in FSP**

Although VDM–SL has syntax for modification, this specification has no explicit modification, and there is no explicit state changes. We perform a kind of dataflow analysis to extract a state machine embodied in this specification. These functions do not have modifications, and do not construct an explicit state model. However, the pair of input with precondition and output of the `Light` type value can be used to construct a state model with actions on the value.

| function | input | output |
|---|---|---|
| ToGreen | Red:Light | Green:Light |
| ToAmber | Green:Light | Amber:Light |
| ToRed | Amber:Light | Red:Light |

Table 1: Functions and their input/output

Table1 shows the result of analyzing specific input and output of these function. Now, we can see a chain of a dataflow of a value of the `Light` type as shown in Figure2. The execution of these functions are triggered by input argument and terminate after returning the rerun value. When we see these functions as an action in LTS, the precondition of the value indicates the postcondition at the source of the action, and the return value indicates the target of the action. By substituting a function with an action, and a specific value with a state in Table1, we can get Table2.

Now we can extract an LTS by the tool LTSA as shown in Figure 3 which can be written as follows in FSP.

```
TRAFFIC = (toGreen -> toAmber -> toRed ->
```

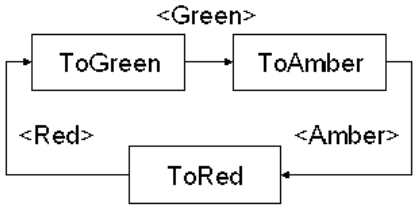


Figure 2: Chain of action (function) on light value

| action | source state | target state |
|---|---|---|
| ToGreen | state0/Red:Light | state1/Green:Light |
| ToAmber | state1/Green:Light | state2/Amber:Light |
| ToRed | state2/Amber:Light | state0/Red:Light |

Table 2: Actions and their source/target

```
TRAFFIC).
```

**Extraction of model in PROMELA**

As explained before, we tried to describe a PROMELA model from the same VDM-SL specification of traffic signal. We decided FSP is more suitable than PROMELA for our purpose, we discuss the PROMELA model for the comparison purpose. The following is the PROMELA model we got from the same VDM-SL specification as before.

```
/* type definitions */
mtype = { RED, AMBER, GREEN };

/* global variables */
mtype light;

/* message channels */
chan signal_msg = [1] of { mtype };

/* inline definition */

inline ToGreen(l)
{
  signal_msg?GREEN;
  l = GREEN
}

inline ToAmber(l)
{
  signal_msg?AMBER;
  l = AMBER
}
```

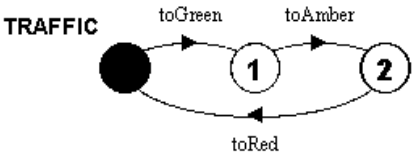


Figure 3: TRAFFIC process

```
inline ToRed(l)
{
  signal_msg?RED;
  l = RED
}


/* process type */

proctype Signal()
{
  do
    ::if
      :: atomic { signal_msg?[GREEN] -> ToGreen(light)
                  signal_msg!AMBER }
      :: atomic { signal_msg?[AMBER] -> ToAmber(light)
                  signal_msg!RED }
      :: atomic { signal_msg?[RED] -> ToRed(light)
                  signal_msg!GREEN }
      fi
  od
}

init
{
  signal_msg!GREEN;
  run Signal()
}
```

”Proctype” is a declaration of process type required before instantiation. Instantiation can be done with operators such as ”run” at the time of declaration. In this model, ”Signal” process is executed in ”init” process, a process executed at first. And a part of process to be executed without interruption is inserted in a curly brackets, after a declaration ”atomic”.

Conversion between VDM-SL specification and PROMELA model is as follows. First, we converted type definitions in the VDM-SL specification to those in PROMELA. The value of each quote types of type ”Light” was related to ”mtype”. ”Mtype” is one of types used in PROMELA. The values of ”mtype” are symbolic names of constant values which is defined by the person who describe the model. In PROMELA, types we can use are similar to those of C. When we describe a model, we must define types of state variables. But the variety of types in PROMELA are less than those in VDM-SL. It is difficult to describe the types like a quote type of VDM-SL specification by using PROMELA.

Next we described the functions in VDM-SL specification by using ”inline definition” in PROMELA. An ”inline definition” is like a procedure in PROMELA. It works much like preprocessor macro. For example, the function ”ToGreen” in VDM-SL specification is described as ”inline ToGreen” in PROMELA.

But we have problems when using PROMELA. When we conduct model checking at an analysis phase, we want to construct an abstract model from a specification in VDM–SL, and examine the behavior of the model which reflects our requirement specification in VDM-SL. In the case of PROMELA, we have to describe detailed type definition of state variables like C, but it is not preferable. Making concrete type definitions may be useful in implementation phase, but cumbersome in analysis phase.

The order of execution of the functions and operations is not described explicitly in VDM-SL specifications. We can guess the execution order by analyzing functions, operations, and their pre-condition and post-condition in the specification. However, when we describe a model in PROMELA, we usually describe the behavior of the model explicitly by message passing. So when we extract a PROMELA model from a VDM-SL specification, we must decide the execution order described in the specification implicitly, and describe such an execution flow in an explicit message passing style. In order to describes the PROMELA model, we must describe when and how messages should be sent and received. This may be possible but is cumbersome and is not suitable for our purpose.

Our goal is to establish a systematic method to construct abstract state models from VDM-SL specifications at an analysis phase of system development process. The problems discussed above hinder our approach. Although more dedicated work may help us to solve these problems, it will need a lot of time. From the reason explained above, we decided to use another language, FSP, which can describe models in abstract and simple notation compared to PROMELA.

## EXTRACTION OF COMPOSED MODEL

We consider a more complex traffic signal system[3], which need process composition. As traffic light control is safety critical system and in this section we consider a safety kernel of the system using the previous simple example. The safety kernel provides the only means of control over the critical functions of the system. Since the kernel is responsible for maintaining safety, it is worth modeling it at an abstract level at an early stage of development, so that the designers have a clear understanding of how the kernel should behave.

Each traffic light at an intersection is responsible for informing drivers, modeled as in the previous example, whether they can drive through the intersection or whether they must wait. A green light is used to indicate that drivers can continue and a red light is used to indicate that drivers must stop. The amber light is used to indicate that the lights are about to change from green to red. A traffic light controller is responsible for controlling the lights. The safety kernel for such a controller should ensure that the light do not permit traffic to flow in conflicting paths simultaneously. A light associ-

ated with a particular path may only change according to the transition shown above.

Basically, we consider the same example as [3], but we omit the timing constraint for simplicity. We focus on the following requirement for the safety kernel of the traffic light controller.

> It must always be the case that if a pair of paths conflict then the lights associated with one of the path is red.

We have four traffic signals for paths, `A1North`, `A1South`, `A66East`, and `A66West`. We have to integrate four traffic signals to construct a system for an intersection.

```
01: values
02:
03:   p1 : Path = mk_token("A1North");
04:   p2 : Path = mk_token("A1South");
05:   p3 : Path = mk_token("A66East");
06:   p4 : Path = mk_token("A66West");
07:
08:   lights : map Path to Light
09:          = {p1 |-> <Red>,
10:             p2 |-> <Red>,
11:             p3 |-> <Green>,
12:             p4 |-> <Green>};
13:
14:   conflicts : set of Conflict
15:             = {mk_Conflict(p1,p3),
16:                mk_Conflict(p1,p4),
17:                mk_Conflict(p2,p3),
18:                mk_Conflict(p2,p4),
19:                mk_Conflict(p3,p1),
20:                mk_Conflict(p4,p1),
21:                mk_Conflict(p3,p2),
22:                mk_Conflict(p4,p2)};
23:
24:   kernel : Kernel
25:          = mk_Kernel(lights,conflicts)
26:
27: types
28:
29:   Light = <Red> | <Amber> | <Green>;
30:
31:   Time = real
32:   inv time == time >= 0;
33:
34:   Path = token;
35:
36:   Conflict :: path1: Path
37:               path2: Path
38:   inv mk_Conflict(p1,p2) == p1 <> p2;
39:
40:   Kernel :: lights    : map Path to Light
41:             conflicts : set of Conflict
42:   inv mk_Kernel(ls,cs) ==
43:         forall conflicts in set cs &
44:              mk_Conflict(conflicts.path2,
45:                          conflicts.path1)
46:              in set cs and
47:              conflicts.path1 in set dom ls and
48:              conflicts.path2 in set dom ls and
49:              (ls(conflicts.path1) = <Red> or
50:               ls(conflicts.path2) = <Red>)
51:
52: functions
53:
54:   ToGreen: Path * Kernel -> Kernel
55:   ToGreen(p, mk_Kernel(lights,conflicts)) ==
56:     mk_Kernel(ChgLight(lights,p,<Green>),
57:                    conflicts)
58:   pre p in set dom lights and
59:       lights(p) = <Red> and
60:       forall mk_Conflict(p1,p2)
61:              in set conflicts &
62:              (p = p1 =>
63:                   lights(p2) = <Red>);
64:
65:   ToRed: Path * Kernel -> Kernel
66:   ToRed(p, mk_Kernel(lights,conflicts)) ==
67:     mk_Kernel(ChgLight(lights,p,<Red>),
68:                    conflicts)
69:   pre p in set dom lights and
70:       lights(p) = <Amber>;
71:
72:   ToAmber: Path * Kernel -> Kernel
73:   ToAmber(p, mk_Kernel(lights,conflicts)) ==
74:     mk_Kernel(ChgLight(lights,p,<Amber>),
75:                    conflicts)
76:   pre p in set dom lights and
77:       lights(p) = <Green>;
78:
79:   ChgLight: (map Path to Light) * Path *
80:             Light -> (map Path to Light)
81:   ChgLight(ls,p,colour) ==
82:     ls ++ {p |-> colour}
```

Since two traffic signals at the opposite side of the same road must be the same and our main focus is conflict, such two traffic lights can be be represented as one state machine.

First we look for explicit modification in this specification. There is an explicit modification `ls ++ {p |-> colour}` in the function `Chglight` (line 82). This modifies the mapping `ls` so that in the range of the mapping the associated value with the value `p` in the domain of the mapping is set to `colour`. Thus, the function `Chglight` modifies its first argument using its second and third argument and returns the modified value as its result. Though we can see this modification as a state change, the specific value is parameterized and we cannot make a state machine only from `Chglight`.

Next we examine the usage of `Chglight` to see how this function is instantiated with actual values. Three functions `ToGreen` (line 54), `ToRed` (line 65) and `ToAmber` (line 72) use `Chglight` as `ChgLight(lights,p,<Green>)` (line 56), `ChgLight(lights,p,<Amber>)` (line 67), and `ChgLight(lights,p,<Amber>)` (line 74), respectively. The return value of these functions is the `Kernel` type value (line 40) and the value of the second argument of these functions except that its first field is modified by `Chglight`. This modification needs to consume `lights` and `p`, and these functions have a precondition concerning `lights` and `p`. Table 3 shows their precondition and modification on `lights(p)` for each function.

Now, we can see a chain of the value of `lights(p)` similar to Figure2, but the chain is parameterized by `p`. When we see these functions as an action in LTS, the precondition indicates the modification result at the source of the action, and the modification result indicates the target of the action. By substituting a function in Table 3 with an action, and also a specific mapping value with a state, we can get Table4.

| function | precondition | modification |
|---|---|---|
| ToGreen | lights(p)=Red | lights(p)=Green |
| ToAmber | lights(p)=Green | lights(p)=Amber |
| ToRed | lights(p)=Amber | lights(p)=Red |

Table 3: Precondition and modification on `lights(p)` for each function.

| action | source state | target state |
|---|---|---|
| ToGreen | state0/lights(p)=Red | state1/lights(p)=Green |
| ToAmber | state1/lights(p)=Green | state2/lights(p)=Amber |
| ToRed | state2/lights(p)=Amber | state0/lights(p)=Red |

Table 4: Actions and their source/target

Please note this state machine is parameterized by `p`. We have additional precondition in the function `ToGreen` (line 60 ... 63) to compose a system from the parameterized state machines. This claims that every path `p2` crossing with `p` must have the mapping `lights(p2)=Red`, or be in the state "state0." Thus `ToGreen` can make the change when —lights(p)=Red— and every path `p2` crossing with `p` has the mapping `lights(p2)=Red` (both `p` and `p2` are in "state0").

In this case it is enough to consider the paths `p1` and `p3` at the top level. The composition is described process labelling by a set of prefix labels in FSP as follows.

```
TRAFFIC = (toGreen -> toAmber -> toRed ->
           TRAFFIC).
RESOURCE = (toGreen -> toRed -> RESOURCE).

||TRAFFICS = (p1 : TRAFFIC || p3 : TRAFFIC
              ||{p1, p3}::RESOURCE).
```

Process prefixing is useful for modelling shared resources between concurrent activities. In this case, we regard "Green" as a shared resource to be acquired exclusively. From the above FSP, we can generate LTS by the tool LTSA as depicted in Figure 4

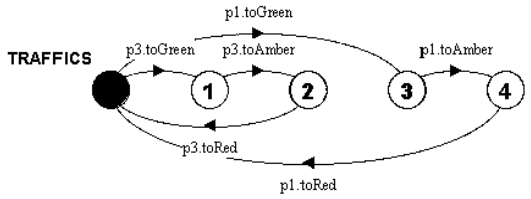


Figure 4: TRAFFICS Process

We can check that the "Green" state is acquired exclusively. However, we can find no control regarding fairness is given in this specification. A transition path like `p1.toGreen`, `p1.toAmber`, `p1.toRed` and `p1.toGreen` is legal in this specification. Some control regarding fairness should be given in the following development.

## CONCLUSION

In this paper, we discussed our approach to extract state machines written in FSP, a notation of a labeled transition system, from specifications in VDM–SL as a part of our multi-aspect system analysis. After trying to generate state machines in PROMELA, a language for a model checking tool SPIN, we found FSP is more suitable for our purpose. We can analyze both static and dynamic aspect by using these two different kinds of specification languages, model-oriented and state-machine languages in an integrated manner.

## REFERENCES


1. J. M. Spivey. "The Z Notation: A Reference Manual Second Edition." Prentice Hall, 1992.
2. Jonathan Bowen. "Virtual Library: Z notation." http://hello.to/zed/
3. John Fitzgerald, and Peter Gorm Larsen. "Modelling Systems." Cambridge University Press, 1998.
4. John Fitzgerald. "Information on VDM." http://www.csr.ncl.ac.uk/vdm/
5. "The IFAD VDM Tools." http://www.ifad.dk/
6. E. M. Clarke, Orna Grumberg and Doron Peled "Model Checking," MIT Press, 2000.
7. Gerard J. Holzmann "The SPIN Model Checker: Primer and Reference Manual," Addison Wesley, 2004
8. "LTSA - Labelled Transition System Analyser," http://www.doc.ic.ac.uk/ltsa/.
9. Jeff Magee and Jeff Kramer "Concurrency: State Models & Java Programs," John Wiley & Sons, 1999.
10. "UML(TM) Resource Page," http://www.uml.org/
11. Bruce Douglass. "Real-Time UML: Developing Efficient Objects for Embedded Systems Second Edition." Addison-Wesley, 2000.
12. Harel, David. "Statecharts: a Visual Formalism for Complex Systems." Science of Computer Programming 8, 1987, 231-274.
13. Bandera Project, http://bandera.projects.cis.ksu.edu/.