Aspect-Oriented Software Design with a Variant of UML/STD

Shin NAKAJIMA
National Institute of Informatics
nkjm@nii.ac.jp

Tetsuo TAMAI
The University of Tokyo
tamai@acm.org

ABSTRACT

The notion of aspect is important as a systematic approach to the representation of cross-cutting concerns and the incremental additions of new functionalities to an existing system. Since UML is a modeling language used in early stages of software development, studying how UML is related to aspectual software is an important topic. This paper proposes a way of introducing the join point model (JPM) to UML/STD. The proposed extension is smoothly integrated with the core part of the execution semantics adapted by the UML standard.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Design

Keywords

UML State Diagrams, Join Point Model, SPIN

1. INTRODUCTION

The notion of aspect is important as a systematic approach to the representation of cross-cutting concerns and the incremental additions of new functionalities to an existing system. Given a base description having certain functionalities, an aspect is added to or weaved into the base in order to create a desired system. Starting as a technology for programming, the aspectual approach is now recognized to be effective as a modeling technology [6].

In the early stages of software development, UML [19] is used as the modeling language and introducing the notion of aspect into UML turns out to be important. UML is a family of notations, each of which can represent a system's certain viewpoint. How the notion of aspect is introduced is dependent on the viewpoint that the diagram provides. Moreover, a complete software design, either aspectual or non-aspectual, requires using various UML notations. There actually have been lots of work to introduce the aspectual concept into UML [2][4][5][6][7]. Weaving is a kind of *model transformation* to have a whole integrated description.

In UML, a state-transition diagram (STD) is used to represent the dynamic behavior of the system. Since operational meanings are important for the dynamic behavior, UML/STD is also accompanied with a set of rules to *execute* the model descriptions. Concequently, the notion of aspect and the weaving is expected to refer to the fine-grained operational meanings or the execution mechanism as was done for the case of the aspect-oriented programming.

This paper proposes an aspectual UML/STD. It defines a join point model based on behavioral specifications of UML/STD and shows that the operational rules can be integrated with the core part of UML/STD's operational semantics [19]. The proposed operational rules are rigorous enough to realize weaving automatically, and can be a basis for the behavioral analysis with SPIN model-checker [9].

2. BACKGROUNDS

Aspect-oriented software development is a technology concerned with both modeling and mechanism.

A primary aim of modeling is to have a clear and easy-to-understand system descriptions at an adequate abstract level. The aspectual notion can help identifying appropriate *concerns*. I.Jacobson [10] adopts his "usecase" modeling approach to represent the aspectual use-cases as well. Theme/UML [2] is a modeling method for mining the aspect and the base, and uses an extended UML representing the aspectual concepts with certain stereotypes.

A mechanism, on the other hand, is related to the computational model of the language to describe the aspectual software. An aspectual language has constructs to describe the aspect as well as the base, and it is equipped with a certain tool to *weave* the aspect into the base. An aspectual mechanism has been mainly studied in the development of aspect-oriented programming languages [15]. AspectJ [12][18] extends Java language to include a language construct (`aspect`) for the representation. An AspectJ compiler then automatically performs the weaving.

In the early stages of software development, UML [19] is used as the modeling language and introducing the notion of aspect into UML turns out to be important. Among the

SCESM'06 May 27, 2006, Shanghai, China

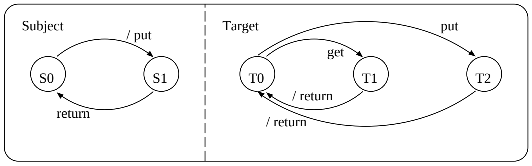

Figure 1: Base

diagrms UML provides, a state-transition diagram (STD) is used to represent the dynamic behavior of the system. Since the operational meanings are important for the dynamic behavior, UML/STD is also accompanied with a set of rules to *execute* the model descriptions. Concequently, the notion of aspect and the weaving is expected to refer to the execution mechanism as was done for the case of AOP. Further, the aspectual UML/STD should be integrated with the core part of UML/STD's operational semantics [19], and be a basis for the behavioral analysis with the model-checker.

The approach discussed in this paper is to introduce a join point model based on the behavioral specifications of UML/STD Join Point Model (JPM) is the basic model adapted in AspectJ, a representative of AOP languages [12][15][18]. Its aspectual notion is captured by the use of advice and pointcuts to define behavior that crosscuts the structure of the base program. *Join points* are a collection of points in execution at which the advice may be executed. Such join points are called a *pointcut*. An *advice* is an action performed at the join points in a particular pointcut.

Since JPM is a typical model to embody the characteristics of the aspectual mechanism, the following sections will study how to introduce a similar model to UML/STD.

3. UML/STD AND JPM

3.1 UML/STD

A UML state diagram (UML/STD) is a hierarchical finite-state transition system based on the Statecharts proposed by D.Harel [8]. It is hierarchical in that a state can have multiple sub-states, and two kinds of the hierarchies are defined, an *And*-hierarchy and an *Or*-hierarchy. A state with an *And*-hierarchy can have multiple sub-states at a time, and those sub-states are considered to execute *concurrently*.

Figure 1 is a simple example of UML/STD, which will be hereafter weaved with an aspect in Section 4. The top-level STD `System` is expanded into an *And*-hierarchy consisting of `Subject` and `Target`. Each is further expanded into an *Or*-hierarchy. `Subject` contains two terminal states, while `Target` is decomposed into three. Intuitively, `System` consists of two component state machines executing concurrently (`Subject` and `Target`). The progress of the state machines is determined by events to fire the transitions.

UML/STD [19] is a large language providing a lot of interesting features, and the standard document describes the syntax and the rules for the execution informally. This paper focuses on the core features of UML/STD; the hierarchical state machine and the RTC (run-to-completion) execution rule. The following discussion will be based on the formalism in [13]. The key idea is to use a configuration term to represent UML/STD formally.

First, a *configuration* is introduced to represent an execution snapshot of a given UML/STD. It is a term representation of the And-Or tree structure of the state hierarchy. For example, the initial state of the example in Figure 1 can be described as a term

```
System(Subject(S0), Target(T0)).
```

Note that two concurrently executing *And*-component machines are explicitly represented as sub-terms of `System`.

Second, a transition is a rewriting rule on the configuration term that is triggered by an appropriate event. Two of the transitions in the example are

```
System(Subject(?X), Target(T0))
  -(get)-> System(Subject(?X), Target(T1))
System(Subject(?X), Target(T1))
  -(/ return)-> System(Subject(?X), Target(T0))
```

where `?X` denotes $don'tcare$, which can be matched with any possible sub-term (either `S0` or `S1` in the example). Intuitively, `Target`'s transition from `T0` to `T1` can be fired regardless of the `Subject` state since two *And*-component machines are considered to execute concurrently.

The following triple, a configuration automaton, is sufficient to represent the formal model of a state machine for the subset of UML/STD in this paper.

(State, Event, Rule)

State : a finite set of (ground) configuration terms

Event : a finite set of events

Rule : a finite set of rewriting rules

Note that a state is a ground configuration term not containing variables and that a rewriting rule is defined on a configuration term that may have a variable such as `?X` as shown in the above example.

The central part of the UML/STD semantics is a RTC step in which a pool (EventPool) is assumed to contain a set of events to be dispatched. The RTC step describes that some of the events in the pool are processed at a time, and that the current events are completely executed before the next set of events is dispatched. Furthermore, a dispatched event that does not trigger any transition is lost (an implicit consumption of an event). An event can also contribute to trigger more than one transitions (a broadcast event).

An execution snapshot is described by a pair consisting of the configuration term and the set of events in the event pool.

<Configuration, EventPool>

The pair is updated according to the RTC step rules. The traces to show how the pair is changed is considered to be *executions* of STD.

Behavioral specification is a set of execution paths that a configuration automaton generates according to the RTC step. The notion of execution paths is defined in terms of a *run*, which is an infinite sequence (π) such that

$$\pi = s_0 s_1 \ldots s_n \ldots,$$

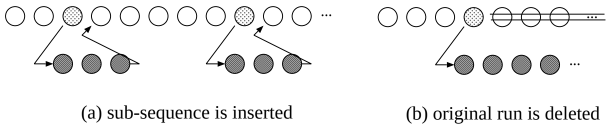

Figure 2: Modifying Run

where s_{k+1} is a configuration term obtained from s_k in the RTC step. Although π is infinite, it can represent a finite run as well by applying the stutter extension rule [9]: namely, a null self transition (ϵ) is added on the *final* state of the configuration automaton.

It is sometimes useful to extend the *run* to include the EventPool since a stable state of the RTC step is a snapshot consisting of the pair. Such an extended run ($\widehat{\pi}$) is defined as

$$\widehat{\pi} = \langle s_0, \xi_0 \rangle \langle s_1, \xi_1 \rangle \ldots \langle s_n, \xi_n \rangle \ldots$$

where ξ_k denotes an EventPool state. A set of extended runs $\widehat{\pi}$ that a configuration automaton generates is written as $\widehat{\Pi}$, which constitutes the behavioral specification of a given UML/STD.

3.2 JPM and Aspect

This paper proposes to use Join point as the basis for introducing the notion of aspect into UML/STD. Since a join point is a certain *point* in the execution sequences, the element of the run ($\langle s_n, \xi_n \rangle$) would be an appropriate candidate. Moreover, a pointcut is a condition that specifies a set of join points. It is an expression for denoting a set of particular elements of the run.

The advice is the action invoked on the join point that a certain pointcut determines. From the viewpoint of the behavioral specifications, the advice results in changes in the behavior of the base state machine. When the behavior is determined in terms of the *run*, i.e. a sequence of locations, the change can be schematically illustrated as in Figure 2. The circle, the location, in the figure refers to an extended element ($\langle s_n, \xi_n \rangle$).

Figure 2 (a) is a case in that a hypothetical *run* of the base would be modified to include a certain sub-sequence at a location specified by the pointcut. The light-gray location is the one specified by the pointcut and a sequence of dark-gray locations are inserted which is the *effect* of the advice.

There may be also a case as shown in Figure 2 (b). The advice results in deleting all the future sequences starting at the location specified by the pointcut. It corresponds to a situation where the behavior of the base is completely changed by the aspect.

The machinery to have such effects as to modify a hypothetical run is certainly attributed to the events that enable particular state transitions. This is because the state transition is the only mechanism responsible for the *execution* of STD. For example, a new event generated at the pointcut location may lead to a completely different execution sequence (Figure 2 (b)).

It is, however, not possible with the state transition alone to have effects on the run as shown in Figure 2 (a). The base is interrupted at the pointcut location (the light-gray location) and suspends itself. Then, a sub-sequence is inserted, which did not appear in the original run, but is an effect by the advice. After the sub-sequence is completed, the base resumes its execution from the point that was interrupted. Consequently, a certain new mechanism is needed to realize the suspending and resuming.

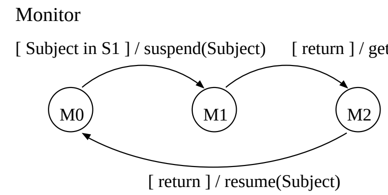

Figure 3: Aspect

4. ASPECT-ORIENTED STATE-DIAGRAM

4.1 Aspectual Example

The base UML/STD in Figure 1 is used as an example for introducing the aspectual machine in Figure 3. The example is meant to show the motivation to introduce the pointcut and the progress control mechanism explained in the rest of this section.

In Figure 1, `Subject` component makes an access to `Target` by generating a `put` event. `Subject` then changes its own state by a `return` event from `Target` to return to the initial state (`S0`). Although `Target` has a transition arc enabled by a `get` event, it is never triggered at all in this base description.

Figure 3 is an aspectual component `Monitor` that introduces a new behavior to the base. The new behavior is the one such that both `put` and `get` events are generated. Two events are generated in this order from the viewpoint of `Target` even if `Subject` generates an `put` event only.

`Monitor` moves from `M0` to `M1` when the `Subject` is in the state `S1`, and it suspends `Subject` by using a primitive control command `suspend(Subject)`. Then, `Monitor` moves to `M2` when an `return` event is inserted into the event pool. Actually the `return` event is generated by `Target`. Last, `Monitor` goes back to `M0` in a similar manner to resume `Subject`.

To study the runs, a handy notation σ_{ij} is introduced.

$$\sigma_{ij} \doteq \texttt{System(Subject(S}i\texttt{),Target(T}j\texttt{))}$$

A snapshot in the extended run is a tuple where $\{\ e_k\ \}$ stands for the event pool. Therefore, a snapshot is represented as

$$\langle \sigma_{ij}, \{\ e_k\ \} \rangle.$$

A typical example behavior of the base in Figure 1 would be written as π_{base}.

$$\pi_{base} \doteq \langle \sigma_{00}, \{\ \} \rangle\ \langle \sigma_{10}, \{\ put\ \} \rangle\ \langle \sigma_{12}, \{\ \} \rangle\ \langle \sigma_{10}, \{\ return\ \} \rangle\ \langle \sigma_{00}, \{\ \} \rangle\ \langle \sigma_{10}, \{\ put\ \} \rangle\ \ldots$$

A new run that is affected by the aspect in Figure 3 is changed to be $\pi_{modified}$ as given below.

$$\pi_{modified} \doteq\ \langle \sigma_{00}, \{\ \} \rangle\ \langle \sigma_{10}, \{\ put\ \} \rangle\ \langle \sigma_{12}, \{\ \ \} \rangle\ \underline{\langle \sigma_{10}, \{\ return,\ get \} \rangle}\ \underline{\langle \sigma_{11}, \{\ \ \} \rangle}\ \langle \sigma_{10}, \{\ return \} \rangle\ \langle \sigma_{00}, \{\ \} \rangle\ \ldots$$

The difference between π_{base} and $\pi_{modified}$ can be seen in the underlined snapshots.

The aspectual machine in Figure 3 always monitors the execution of the base, actually the *run* generated by the base. More concretely, the machine watches all the information contained in the tuple $(\langle s_n, \xi_n \rangle)$. A short sub-sequence

$$(\langle \sigma_{10}, \{return,\ get\} \rangle \langle \sigma_{11}, \{\ \} \rangle)$$

is inserted into a certain location in π_{base}. This is an example of the schematic diagram in Figure 2 (a).

4.2 Pointcut

The pointcut is a means to express conditions to select certain join points. The join point here is $\langle s_k, \xi_k \rangle$ where s_k is a (ground) configuration term and ξ_k is an event pool. Since a pointcut is an expression for discriminating join points uniquely, it is enough to include two kinds of atomic propositions: one posed on s_k and another on ξ_k. Here is an abstract syntax of the pointcut.

$$\mathrm{P} := \mathrm{M\ in\ S} \mid \mathrm{E} \mid \neg P \mid P \wedge P \mid P \vee P$$

The first atomic proposition (M in S) is *true* when a component state-machine M is in the state S. The second one (E) is *true* when the event pool contains the specified event.

As for the example in Figure 3, the transition from `M0` to `M1` is triggered when `Subject` is in the state `S1`: namely, the configuration term takes a form of:

```
System(Subject(S1),Target(?X))
```

The transition from `M1` to `M2` is enabled when a `return` event is generated and inserted into the event pool. The monitored event (`return` in this case) is not consumed but it will be dispatched in the next RTC step.

In order to deal with the pointcut expressions in the RTC-based execution rules of UML/STD, the expressions should be evaluated against a certain stable snapshot of the system $(\langle s_n, \xi_n \rangle)$. An RTC step consists of lots of micro-steps and the intermediate states are not stable nor well-defined. The pointcut is evaluated against the system status at the end of each RTC step.

4.3 Controlling Progress of Machine

As discussed in Section 3.2, a certain machinery to control the execution of a STD is needed to have the modified sequences as in Figure 2. A primitive, a `provided` clause, to control such executions is introduced. A `provided` clause takes a form as below in which `N` refers to a name of a component state-machine and `P` describes the condition.

```
N provided P
```

Operationally, the sub-machine `N` is *scheduled* only when the condition `P` is *true*. Here, being *scheduled* refers to the situation that transitions defined on `N` are consulted when events are dispatched. Conversely, when `N` is not *scheduled* because of `P`'s *false* value, no transition occurs on `N` even if appropriate events to fire the transition are dispatched.

The condition `P` takes either of the following.

$$P := \mathrm{M\ in\ S} \mid false \mid true \mid \neg P \mid P \wedge P \mid P \vee P$$

A simple example might be adequate here: `Example` consists of several *And*-components including `N` and `M`, and `N` has a provided clause of `(M in S1)`.

```
Example(..., N(T), ..., M(S), ...)
```

The *And*-component machine named `N` is *scheduled* when its brother machine `M` is in the state `S1`. `M` can thus control the progress of `N`.

Since a `provided` clause is introduced to control over the progress of the component state-machines, this notion should be included in the formal definition of UML/STD. Thus, the configuration automaton becomes a quadruple

(State, Event, Rule, ProvidedClause)

where ProvidedClause is a mapping from the name of the component state-machine to its `provided` clause.

Moreover, the binding relationship can be changed in the course of the execution.

```
provided(N) := P
```

Such a modification primitive can be a part of the action taken when a certain transition occurs. By using this primitive, two progress control actions used in Figure 3 are actually *macros*.

```
suspend(N) ≐ provided(N) := false
resume(N)  ≐ provided(N) := true
```

By using these primitives, `Monitor`'s action is that (1) suspending the execution of `Subject` at a transition from `M0` to `M1`, (2) generating `get` event at a transition from `M1` to `M2` when first `return` is generated, and (3) resuming `Subject` when another `return` comes.

5. FORMAL ANALYSIS

5.1 Model-Checking

Model-checking is an analysis method for dynamic behaviors represented by UML/STD. Its basic idea is to explore all the state space that a given UML/STD design constructs, and to decide whether a given property holds or not. There has been a study on model-checking UML/STD [13][16][17]. This paper adapts similar techniques for the formal analysis of the aspectual UML/STD. However, as discussed in Section 3, behavioral specification of UML/STD is defined in terms of run (a sequence of $\langle s_n, \xi_n \rangle$). The verification problem will be presented here in terms of the run.

For a given UML/STD description to generate a run $(\widehat{\pi})$, the generation method should follow the rule defined in the RTC step being modified to take into account the pointcut evaluation. Such a design description may, in general, have non-determinism and thus generate a set of runs $(\widehat{\Pi})$. A state space referred to above is actually a graph (a set of nodes and a set of edges)[1] for representing the set compactly. A node of the graph is a tuple to express the snapshot of the configuration machine status, and an edge denotes that connecting two nodes are adjoining in a run.

Next, if a logical formula f is assumed to stand for a property to be checked, the verification problem is defined as to test whether f is valid or not with respect to the set of runs.

$$\widehat{\Pi} \models \mathrm{f}$$

[1]It is called Kripke Structure in literatures.

As for expressing temporal properties, LTL (linear temporal logic) is a convenient means and is used to query whether a state machine shows a particular behavior or not.

An LTL formula *f* can have three temporal operators, `[]` (always), `<>` (eventually), and `U` (strong until) as well as usual logical connectives such as `!` ($\neg$), `&&` ($\wedge$), `||` ($\vee$), and `->` ($\Rightarrow$). Their semantics follow the standard definitions [9] and are skipped here. Atomic propositions, referred to in the formula, are chosen so as to be defined on the snapshot of the pair $\langle s_k, \xi_k \rangle$.

- M in S : referring to configuration term s_k
- E ($\in$ Event) : referring to event pool ξ_k

Their meanings are the same as in the case of the pointcut in Section 4.2.

In regard to conducting the analysis, the SPIN model checker [9] is used. This is because SPIN is quite an efficient model-checker and it is often used as a back-end engine for model-checking of design notations such as UML/STD and programs such as C or Java.

Using SPIN for the model-checking engine for the aspectual UML/STD is straight forward. A given UML/STD design description, actually its configuration machine, is translated into Promela, that is the input specification language of SPIN. Since the translated Promela program should be faithful to the core semantics of UML/STD and the pointcut evaluation, it consults the micro-steps in the modified RTC cycle. This can be done by the Promela program, which incorporates an *interpreter* to implement the modified RTC steps together with the surface description of the given UML/STD. Moreover, the Promela program can be written so as to have non-deterministic choices of many transitions; thus introducing non-determinism is easy.

5.2 Weaving and LTL Formulas

It is important to ensure that the design after weaving hold a certain required properties, and LTL formulas are adequate means for this purpose.

Some simple example LTL formulas are shown below for the case of the design description in Figure 1 and 3.

First, the base design in Figure 1 satisfies

```
[](put -> <> return) ... (a)
```

which reads such that it is always the case that a `put` event is eventually followed by a `return`. The formula is also valid after the base is weaved into the aspect in Figure 3.

Second, there is an LTL formula which the base satisfies but is not valid after weaving. For example, an LTL formula, meaning that it is always the case that a `put` event is eventually generated, but a `get` event is never generated, is written as

```
[]<>put && !([]<>get) ... (b)
```

After weaving, it is not satisfied any more. Contrarily, the LTL formulas below are valid.

```
[]<>put && []<>get
[](put -> <>get)
```

These formulas are actually the required properties to hold and the formula (b) is the one to be avoided. By checking an appropriate LTL formulas, the weaved design can be shown to satisfy the requirements.

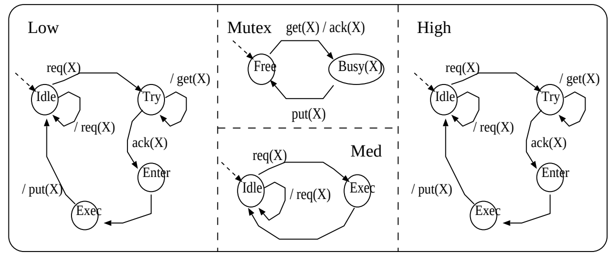

Figure 4: Priority Inversion

6. AN APPLICATION

6.1 Priority Inversion Phenomena

The problem at hand considers scheduling tasks with mutually exclusive dependencies. There are three tasks that are given execution priorities just as their names suggest. Two of them, `Low` and `High` have a shared resource `Mutex`, while `Med` can execute freely in a sense. A faulty behavior may appear when `Low` locks `Mutex` while `High` waits for its release. At a certain scheduling point, `Med` is executable because the priority of `Med` is higher than `Low`. Note that `High` is not runnable because it waits for the release of `Mutex`. Then it shows the *Priority Inversion Phenomena* in that `High` is blocked while `Med` is in execution [3].

The primary role in the example is the notion of priority. It actually controls the progress of a component state-machine, which can be represented easily with the `provided` clause. Figure 4 is a UML/STD description for the present situation. In addition to the behavior described with the diagrammatic notation, a task state-machine with a differing execution priority is accompanied with an appropriate `provided` clause.

```
Med provided  !(High in Exec)||(High in Enter)
Low provided !((High in Exec)||(High in Enter))
          && !(Med in Exec)
```

The above specifies that `Med` can proceed only when the specified condition is satisfied: `High` is not in `Exec` nor in `Enter` which corresponds to the situation that `High` is not in execution. The clause for `Low` is dependent on both `Med` and `High` tasks.

Next, the design description in Figure 4 is analyzed by using Promela/SPIN. The verification problem at hand is to check whether the system is free from any of faulty behavior or not. To study such potential faults, a progress property, or a *leads-to* property, expressed in terms of LTL formula, is used.

```
[]((High in Try) -> <>(High in Exec))
```

It reads such that `High` eventually goes to the `Exec` state if it issues a request to obtain the shared resource in the `Try` state.

The result of the model-checking returns a counter example scenario: `Low`, holding the shared resource `Mutex`, is not scheduled at all because `Med` executes forever. The situation is what is called *priority inversion* since `Med` with a priority lower than `High` executes while `High` is blocked.

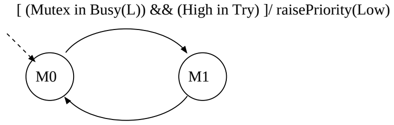

Figure 5: Aspect for Priority Inheritance

6.2 Priority Inheritance Protocol

It is known that the priority inversion phenomena can be resolved by several methods [3]. The *priority inheritance protocol* is introduced to resolve the faulty situation below.

The basic idea is that the priority of `Low` task is temporarily raised to that of `High` waiting for the release of `Mutex`. When `Low` is competing to get scheduled with `Med`, `Low` is chosen because its priority is now higher than `Med`. Since `Low` eventually unlocks `Mutex`, `High` can obtain the shared resource as expected.

Although such a change may usually be scattered in various entities, it is not hard to introduce an aspectual machine responsible for the dynamic priority control. Figure 5 is a representation of such an aspectual machine that is weaved into the base description (Figure 4). The weaving is simply to add the aspectual machine as a new *And*-component state machine.

The aspectual machine, while being in `M0` state, monitors the execution status of the other components. It goes to `M1` when `Mutex` is locked by `Low` and `High` is in `Try` state to try obtaining `Mutex`. The automaton changes the priority of `Low` to that of `High` as an effect of the transition (`raisePriority(Low)`). The monitor automaton, then, put it back the priority of `Low` (`initialPriority(Low)`) when `High` starts its execution with `Mutex` obtained (`Enter` state) and `Low` is in `Idle` state. For simplicity, the function, `raisePriority(Low)`, looks as

```
provided(Low) := true
```

which specifies that `Low` is always scheduled. Although the condition is stronger than the standard priority inheritance protocol, it is suffice for the purpose here to say that the priority of `Low` is higher than that of `Med`.

The whole system can be shown to have no faulty behavior by checking the *leads-to* property for the `High` task by using the LTL formula in Section 6.1.

6.3 Overhead in Model-Checking

To see how much computational overhead is introduced in model-checking the proposed aspectual STD, a simple reachability check is conducted by SPIN.

First, the base design model in Figure 4 is checked. Although it shows a priority inversion phenomenon, the reachability analysis result shows that the system is deadlock free. This is because `Med` task is executed forever, which the system as a whole is not in a deadlock.

As a quantitative comparison to see the overhead introduced by the `provided` clause evaluation, the same system without any `provided` clause is analyzed. Since the system does not have any notion of priority, the reachability analysis for this simplified description is just to test whether there is an inadequate sequence of mutex operations leading to certain deadlock situations. The analysis result demonstrates that no such faulty situation occurs. The size of the state space to be explored is about 25% smaller than the system with `provided` clauses. In another words, about 25% increase in the size of the state space is an overhead relating to the evaluation of the `provided` clause.

Second, the aspectual automaton executes in every RTC cycle to check the status of the other component state machines, which may result in an increase in the size of the state space. The overhead of the aspectual machine is not small, and the state space is about 90% larger than the system shown in Figure 4. This is because the aspectual machine executes in every RTC cycle to check whether a certain pointcut is satisfied or not. The state space inevitably becomes large and is almost double for this example.

All the experiments reported above used the latest version of the SPIN tool version 4.2.5 executed under Windows/XP operating on a laptop computer. All the experiments terminated almost instantaneously, which shows that the translated Promela program did not have any trouble from the viewpoint of the runtime cost.

7. RELATED WORK

This paper adapts the formalism of UML/STD proposed by J.Lilius and I.P.Plator in [13], especially the idea of representing the hierarchical state-machine as a configuration term. And it introduces the join point model and other related aspectual concepts into a variant of UML/STD.

The idea of using the `provided` clause to control the execution of the state-machine at a *meta* level is not new to this work. The `provided` clause in Promela/SPIN [9] can have any condition so that it is very expressive. This work restricts the use of the `provided` clause to have the proposition of the form in Section 4.3. Although restricted, it can be used to describe interesting designs such as the one involving task scheduling. As for the implementation for the model-checking, `provided` of Promela is not used, but its interpretation is integrated with the RTC step of UML/STD. In UML/STD, a similar execution control can be described by a set of guard conditions on transition arcs. It has a drawback, however, in that guard conditions should appropriately be defined on a lots of transitions. On the other hand, only one `provided` clause can represent the condition for the control.

M. Mahoney et al. [14] use the idea of the aspect to have a set of statechart descriptions that are adequately modularized. The key idea is to provide a modeling method in which one or more separate statecharts are woven by using auxiliary declarations of how they are put together. The woven design is used as an input for an automatic generation of executable codes. Hence, their work is related to a model-driven development method. Verification of the statecharts with the aspect annotations is not discussed.

J. Araujo et al. [1] introduce the notion of aspect into the scenario-based software requirement research. A scenario is basically a description of event sequences. Some of the identified scenarios are considered to be the base, while others are aspectual. All the scenarios are merged to synthesize a state machine that generates the event sequences as its behavioral specifications. In their work, the notion of pointcut is implicit in the synthesis algorithm.

E. Katz and S. Katz [11] propose a verification method for checking the validity of the synthesized scenarios. Aspectual scenario has a pointcut in terms of LTL formula.

The method consults the pointcut expression to synthesize a state machine from a given set of scenarios. The obtained state machine is considered as a result of weaving of the given scenarios. However, a scenario is a sequence of *what has happened* and can be divided into several fragments. Some of the fragments are atomic in that they cannot be divided further in the weaving process. The verification problem is to ensure that the state machine does not violate the atomicity that the given scenarios originally have.

As discussed in Section 5.2, the LTL formula is adequate to specify requirements and *coarse-grained* properties. It is not easy to write down the LTL formula to express the condition on the correctness of the weaving in the sense that E. Katz and S. Katz defined. A certain form of representing and verifying the *atomicity* of the base is called for.

8. DISCUSSIONS AND CONCLUSIONS

This paper focused on studying an aspectual mechanism for a UML state diagram (UML/STD) and introduced JPM into UML/STD. In the proposal, weaving was just an addition of an aspect machine and thus was done automatically. The paper further investigated how behavioral analysis of the aspectual design was conducted by using the SPIN model-checker. The proposed *aspectual state diagram* aids understanding of the key notion of the aspect-oriented design in UML/STD.

Last, it is interesting to point out that a certain *aspectual* design can be represented without the proposed method. It is because UML/STD [19] is very expressive and in particular has a notion of *broadcast* event. In the literature on aspectual software, a *logging* aspect is often used. A base system may consists of two components in which `Task` issues `get` event to have an access to `Target`. Adding a logging function to this base can easily be accomplished in UML/STD just by introducing a new *And*-component machine (`Logger`) into the base system. Since an event is broadcast, `Logger` machine as well as `Target` can see the `get` event that `Task` generates. The functionality of `Logger` is ready to be activated without any further machinery.

Although the logging example is simple, the design that does not affect the base can be represented by the *original* UML/STD alone. Namely, it is not necessary to use the proposed *aspectual state diagram*. It is interesting to point out here again that both modeling and mechanism are important, but are two distinct technologies, which was discussed in Section 2.

9. REFERENCES

[1] J. Araujo, J. Whittle, and D. Kim. Modeling and Composing Scenario-Based Requirements with Aspects, In *Proc. RE 2004*, pages 58–67, September 2004.

[2] S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design*. Addison-Wesley 2005.

[3] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Scheduling in Real-time Systems*. Wiley 2002.

[4] M. Deubler, M. Meisinger, S. Rittmann, and I. Kruger. Modeling Crosscutting Services with UML Sequence Diagrams. In *Proc. MoDELS 2005*, pages 522-536, 2005.

[5] T. Elrad, O. Aldawud, and A. Bader. Expressing Aspects Using UML Behavioral and Structural Diagrams. In [6], pages 459–478, 2005.

[6] R. Filman, T. Elrad, S. Clarke, and M. Aksit. *Aspect-Oriented Software Development*. Addison Wesley 2005.

[7] R. France, I. Ray, G. Georg, and S. Ghosh. An Aspect-Oriented Approch to Design Modeling. *IEE Proceedings*, 151 (4), August 2004.

[8] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Trans. Softw. Engin. Meth.*, Vol.5, No.4, pages 293-333, 1996.

[9] G. Holzmann. *The SPIN Model Checker*. Addison-Wesley 2004.

[10] I. Jacobson and P.-W. Ng. *Aspect-Oriented Software Development with Use Cases*. Addison Wesley 2005.

[11] E. Katz and S. Katz. Verifying Scenario-Based Aspect Specifications. In *Proc. FM 2005*, pages 432–447, July 2005.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. ECOOP'97*, 1997.

[13] J. Lilius and I.P. Paltor. The Semantics of UML State Machines. TUCS TR No.273, May 1999.

[14] M. Mahoney, A. Bader, T. Elrad, and O. Aldawud. Using Aspects to Abstract and Modularize Statecharts. UML 2004 Workshop on Aspect-Oriented Modeling, October 2004.

[15] H. Masuhara and G. Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In *Proc. ECOOP 2003*, 2003.

[16] E. Mikk, Y. Lakhnech, M. Siegel, and G. Holzmann. Implementing Statecharts in Promela/SPIN. In *Proc. WIFT'98*, 1998.

[17] T. Schafer, A. Knapp, and S. Merz. Model Checking UML Stata Machines and Collaborations. Electronic Notes in Theoretical Computer Science, Vol.55, No.3, 2001.

[18] M. Wand, G. Kiczales, and C. Dutchyn. A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming. ACM TOPLAS, Vol.26, No.5, pages 890–910, September 2004.

[19] OMG – Unified Modeling Language, v1.5, March 2003.