Rewriting Logic Model of Compositional Abstraction of Aspect-Oriented Software

Yasuyuki Tahara
The University of Electro-Communications, Tokyo, Japan
tahara@is.uec.ac.jp

Akihiko Ohsuga
The University of Electro-Communications, Tokyo, Japan
akihiko@ohsuga.is.uec.ac.jp

Shinichi Honiden
National Institute of Informatics and The University of Tokyo, Japan
honiden@nii.ac.jp

ABSTRACT

Abstraction is an operation of software specifications widely used in formal development and verification. One of the desirable features of the operation is compositionality. It would make abstraction easier to deal with if a system can be abstracted by composing the individual abstractions of the components of the original system. It is considered that compositional abstractions of aspect-oriented software would be useful because the base system and the aspects can be individually abstracted. However, there are only a few research results dealing with these operations consistently because the relation between abstraction and aspect weaving is logically too complicated. This paper proposes a formal model to solve the difficulty of compositional abstraction of aspect-oriented software. Our model is based on an enhanced version of the equational abstraction approach in rewriting logic that is an algebraic specification framework. We first validate our model by applying it to an example of state machine and next describe our approach to compositional abstraction.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*object-oriented design methods*; D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*algebraic approaches to semantics*

General Terms

Design, Theory, Verification

Keywords

Abstraction, State Machines, Rewriting Logic, Compositionality

1. INTRODUCTION

Aspect-oriented software development (AOSD) is spreading quickly and widely these days. It is considered as providing a modularization facility to large-scale practical programs from a direction different from the traditional approaches such as structured, object-oriented, and component-based ones. In AOSD, cross-cutting concerns are extracted from the specifications of both the static structures and the dynamic behaviors and are encapsulated as aspects. In this paper, we focus on behavioral specifications. The behaviors of aspect-oriented software are so complicated that it is not easy to inspect such software manually if it has a practical scale and complexity. This is because formal models and formal verification techniques for aspect-oriented software are desired and many approaches [9, 1, 10, 20, 22, 21, 5, 13, 12, 11] are proposed.

The desirable features of formal models of modularization approaches include facilities of modular or *compositional reasoning and verification*. We mean by the word "compositional" that if we check some property for each module individually we can conclude that the entire system satisfies the same or another property. In this paper, we focus on the abstraction relation between behavioral specifications used in efficient model checking, which is a verification technique. Abstraction means the existence of a mapping between the state spaces of two state-transition behavior models that preserves the transition relations and some properties satisfied by the states. The model of the target of the mapping is called the abstract model. The significance of this relation comes from the following fact. If an abstract model satisfies a property written in some temporal logic, the refined model also satisfies the property. From this viewpoint, any existing approaches are insufficient. Only Jagadeesan et al. [11] deal with the (bi)simulation relation between specifications that is relevant to abstraction in a compositional manner. However, it is not easy to see if two given specifications have the relation or not.

In this paper, we propose a formal model of aspect-oriented software based on an algebraic specification framework called *rewriting logic* [14]. We also present an approach to specify abstraction relations on the basis of the notion of *equational abstraction* [15]. Rewriting logic is a logical framework in which we can derive equality and rewriting relations between terms from a set of axioms called a *rewrite theory* consisting of equations and rewrite rules. As a computation model, a state of a system is represented by an equivalence class of terms with respect to the equality, and transitions are represented by rewriting relations between the terms. Equational

abstraction is a very simple framework to create an abstract model by adding a set of equations. The projection function from the terms representing the system state to their equivalence classes is the abstraction mapping. In our approach, the behaviors of the base system and the aspect are individually modeled by the rewriting relations of the state terms of each system in one rewrite theory. We can create the rewrite theory that models the system in which the aspects are woven into the base system. Then the union of the sets of equations between the base systems and the aspects, respectively, can produce an abstraction mapping between the entire systems in a consistent way. As a result, for example, our approach enables us to carry out abstraction of an aspect-oriented system specifications by abstracting the base system and the aspect specifications individually as shown in Figure 1. This feature would be especially useful for aspect-oriented software because the entire specifications tend to become much larger and much more complicated than the base system and the aspect specifications. Our approach would reduce the cost of abstracting the entire system to the total of the costs of abstracting each components.

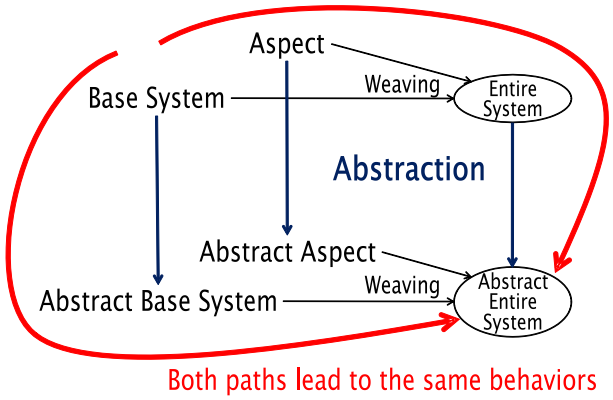

Figure 1: Individual Abstraction of Base System and Aspect

This paper is organized as follows. In Section 2 we summarize the background information of our approach consisting of the state machine model of aspects for compositional model checking and rewriting logic. Section 3 presents our aspect modeling method and justify of our model by demonstrating that we can model an existing formal model correctly. Section 4 describes the equational abstraction in our approach and shows that our model can realize compositional abstraction. Section 5 compares our proposal with the related research efforts. Section 6 presents some concluding remarks and future work.

2. BACKGROUNDS

In this section, we summarize the background information of our approach consisting of the state machine model of aspects for compositional model checking and rewriting logic.

2.1 State Machine Model of Aspects

A state machine M is a tuple $(S, S_0, \rightarrow, L)$ consisting of the set of the states S, the set of the initial states $S_0 \subseteq S$, the transition relation $\rightarrow \subseteq S \times S$, and the labeling function $L : S \rightarrow 2^{AP}$ where AP is the finite[1] set of atomic propositions. We assume that different truth values can be assigned to each atomic proposition in AP at different states in S. For any state s, $L(s)$ is the set of atomic propositions that are true at s. This definition of a state machine corresponds to a Kripke structure in the literature [7] enhanced with the initial states. "$\rightarrow$" must be total, that is, $\forall s \in S\, \exists s' \in S\, (s \rightarrow s')$. For general binary relation $\rightarrow \subseteq S \times S$ that is not total, we can create a total relation $\rightarrow^{\bullet} = \rightarrow \cup \{(s,s) | s \in S, \neg \exists s'\, (s \rightarrow s')\}$ that is an extension of $\rightarrow$.

The following definitions form a simplified version of Katz and Katz [12].

DEFINITION 1. *An* aspect machine *A over a set of atomic propositions AP is a tuple $(S_A, S_0^A, S_{ret}^A, \rightarrow_A, L_A)$ where S_A is the set of the states, $S_0^A \subseteq S_A$ is the set of the initial states, $\rightarrow_A \subseteq S_A \times S_A$ is the transition relation, $L_A : S_A \rightarrow 2^{AP}$ is the labeling function, and $S_{ret}^A \subseteq S_A$ is the set of return states such that $\forall s \in S_{ret}^A \forall s' \in S_A\, (s \rightarrow s'$ implies $s = s')$.*

DEFINITION 2. *A* pointcut descriptor *ρ over a set of atomic propositions AP is a predicate on finite sequences of labels. This means that $\rho(\lambda)$ is a boolean value for each $\lambda = l_0 l_1 \ldots l_n$ where $l_i \subseteq AP\, (i = 0, \ldots, n)$.*

DEFINITION 3. *For a state machine $M = (S, S_0, \rightarrow, L)$ and a (finite or infinite) state sequence $s_0 s_1 \ldots (s_n)$, $label(s_0 s_1 \ldots (s_n))$ is a label sequence $L(s_0)L(s_1)\ldots(L(s_n))$. A pointcut descriptor ρ* matches *a finite state sequence $s_0 s_1 \ldots s_n$ if and only if $\rho(label(s_0 s_1 \ldots s_n))$ is true .*

DEFINITION 4. *Let $B = (S_B, S_0^B, \rightarrow_B, L_B)$ be a state machine over a set of atomic propositions AP_B, ρ be a pointcut descriptor over AP_B, and "pointcut" be a symbol that is not an element of AP_B. Another state machine $B^\rho = (S_{B^\rho}, S_0^{B^\rho}, \rightarrow_{B^\rho}, L_{B^\rho})$ is said to be a* pointcut-ready machine *for B and ρ if and only if the following conditions hold.*

- $S_{B^\rho} \supseteq S_B$
- $L_{B^\rho} : S_{B^\rho} \rightarrow 2^{(AP_B \cup \{pointcut\})}$
- $\forall s_0, \ldots, s_k \in B^\rho\, (s_0 \rightarrow_{B^\rho} s_1 \rightarrow_{B^\rho} \ldots \rightarrow_{B^\rho} s_k$ *and* $s_0 \in S_0^{B^\rho}$ *implies* $(\rho(label(s_0 s_1 \ldots s_k))$ *if and only if pointcut* $\in L_{B^\rho}(s_k)))$
- $\forall l \in (2^{AP_B})^\omega\, ((\exists \pi_{B^\rho} :$ *path in* $B^\rho\, (label(\pi_{B^\rho}) = l))$ *if and only if* $(\exists \pi_B :$ *path in* $B\, (label(\pi_B) = l)))$

  where for a set S, S^ω is the set of infinite sequences of elements of S.

DEFINITION 5. *Suppose the following constructs are given.*

- *An aspect machine $A = (S_A, S_0^A, S_{ret}^A, \rightarrow_A, L_A)$ over AP*
- *A pointcut descriptor ρ over AP*
- *A state machine $B = (S_B, S_0^B, \rightarrow_B, L_B)$ called a* base machine *over $AP_B \supseteq AP$ and its pointcut-ready machine $B^\rho = (S_{B^\rho}, S_0^{B^\rho}, \rightarrow_{B^\rho}, L_{B^\rho})$*

[1] AP may be an infinite set in general. However, we deal with only finite sets as AP in order to enable M to be represented in rewriting logic.

Then we define as follows the augmented machine $\tilde{B} = (S_{\tilde{B}}, S_0^{\tilde{B}}, \rightarrow_{\tilde{B}}, L_{\tilde{B}})$ *in which the aspect machine is woven into the base machine.*

- $S_{\tilde{B}} = S_{B^\rho} \cup S_A$
- $S_0^{\tilde{B}} = S_0^{B^\rho}$
- $\rightarrow_{\tilde{B}} = \{(s,t) \in \rightarrow_{B^\rho} | \mathit{pointcut} \notin L_{B^\rho}(s)\} \cup \rightarrow_A \cup \{(s,t) \in S_{B^\rho} \times S_0^A | \mathit{pointcut} \in L_{B^\rho}(s), L_{B^\rho}(s) \cap AP = L_A(t)\} \cup \{(s,t) \in S_{ret}^A \times S_{B^\rho} | L_A(s) = L_{B^\rho}(t) \cap AP\}$
- $L_{\tilde{B}} = L_{B^\rho} \cup L_A$ *as sets of pairs*

Note that we omitted the treatment of fair states because we do not mention model checking in this paper.

2.2 Rewriting Logic

Next we explain rewriting logic. It can be summarized as follows.

- The most primitive construct of rewriting logic is a *term* that is a syntactic representation of a data or a state. Each term may have *sorts* representing data types.
- A logical formula of rewriting logic is an *equality* relation or a *rewriting* relation between terms. An equivalence class of terms represents a state of a system. A transition between states is represented by the rewriting relation between equivalence classes induced from the one between terms. For example, suppose $[t_1]$ and $[t_2]$ represent two states where $[t]$ denote the equivalence class t belongs to. Then a rewriting relation between terms $t_1 \rightarrow t_2$ induces $[t_1] \rightarrow [t_2]$ representing a transition from $[t_1]$ to $[t_2]$.

A term is composed by symbols for constants, variables, and operators that represent primitive data, placeholders for terms used to express generic equations or rewrite rules, and data structure constructors or operations on data, respectively. A constant symbol is usually treated as an operator symbol with no arguments. For example, $f(a,x)$ is a term if a is a constant symbol, x is a variable symbol, and f is an operator symbol with two arguments. Mixfix operators such as "$+ - \times /$" for numbers can be treated by the placeholder symbol "_". For example, $\_+\_(2,3)$ can also be written as $2+3$ by replacing the two "_"s with the two arguments 2 and 3. A term is said to be *closed* if it includes no variable symbols.

Rewriting logic usually deals with sorts representing data types. It can be decided if a term *has* a sort or not. If a term t has a sort s, we call t a term of the sort s. Although it is recently usual to assign multiple sorts to one term at the same time, we do not deal with such cases in this paper only for simplicity. However, it would be not difficult to extend our approach to such general cases. The assignment of sorts to terms is derived from the initial assignment of sort information to the variable and the operator symbols. As for the example of the term $f(a,x)$ above, if a and x has the sorts s_1 and s_2 respectively and f is defined as an operator producing a term of the sort s from two arguments with the sorts s_1 and s_2, $f(a,x)$ has the sort s. We write these definitions as $a : s_1$, $x : s_2$, and $f : s_1 \times s_2 \rightarrow s$. We also write $\mathrm{Term}(s)$ for the set of the terms having the sort s.

The logical formulae of rewriting logic are equality relations or rewriting relations between terms. They respectively are expressed by the symbols "=" and "$\rightarrow$" and derived from axioms of equality and rewriting relations according to some inference rules. Each type of axioms is called equations and rewrite rules respectively. A (conditional) equation is a logical formula "$t = t'$ if $t_1 = t_1'$, ..., $t_n = t_n'$" where t, t', t_i, and $t_i' (i = 1, \ldots, n)$ are terms. A (conditional) rewrite rule is a formula "$t \rightarrow t'$ if $t_1 \Rightarrow t_1'$, ..., $t_n \Rightarrow t_n'$" where t etc. are terms in the similar way and $\Rightarrow$ denotes $=$ (the equality symbol) or $\rightarrow$ (the rewriting relation symbol). The both sides of each "=" and "$\rightarrow$" need to have the same sort. The inference rules are described as follows.

Reflexivity: For any term t, $t \Rightarrow t$

Symmetry: For any two terms t, t', if $t = t'$, $t' = t$

Congruence: For any operator f, $t_1 \Rightarrow t_1'$, ..., and $t_n \Rightarrow t_n'$ altogether imply $f(t_1, \ldots, t_n) \Rightarrow f(t_1', \ldots t_n')$, where all $\Rightarrow$'s coincide ($=$ or $\rightarrow$).

This rule expresses that each subterm can be rewritten individually.

Replacement: For any axiom "$t(x_1, \ldots, x_n) \Rightarrow t'(x_1, \ldots, x_n)$ if $s_1 \Rightarrow s_1', \ldots, s_n \Rightarrow s_n'$",

$s_1(\bar{w}/\bar{x}) \Rightarrow s_1'(\bar{w}/\bar{x}), \ldots$, and $s_n(\bar{w}/\bar{x}) \Rightarrow s_n'(\bar{w}/\bar{x})$ altogether imply $t(\bar{w}/\bar{x}) \Rightarrow t'(\bar{w}/\bar{x})$, where $\bar{w}$ denotes a sequence of terms $w_1, \ldots, w_n$ and $\bar{w}/\bar{x}$ denotes the componentwise substitution of x_i's to w_i's.

This rule produces relations instantiated from the axiom by substituting the variables $\bar{x}$ to terms $\bar{w}$ if w_i's satisfy the conditions.

Note that the original Replacement rule in the literature such as [14] follows from the above rule, the Congruence rule, and the Transitivity rule.

Equality and Transitivity: $t_1 \Rightarrow t_2$ and $t_2 \Rightarrow t_3$ altogether imply $t_1 \Rightarrow t_3$, where all $\Rightarrow$'s coincide (Transitivity), or one of the $\Rightarrow$'s in the premise is $=$ and the other two are $\rightarrow$'s (Equality). Note that the original Equality rule is derived by applying the above Equality rule twice for both sides of $\rightarrow$.

An axiomatic system of rewriting logic is called a *rewrite theory*. A rewrite theory is a tuple (S, Σ, V, E, R) where each component represents the set of sorts, the signature (the initial assignments of the operator symbols to their sort information), the initial assignments of the variable symbols to the sorts, the set of equations, and the set of rewrite rules, respectively.

In order to describe concrete rewrite theories, we use the notation of the Maude language [8] in which sorts, signatures, variable symbols, equations, and rewrite rules are declared with the keywords `sort`, `op`, `var`, `eq`, and `rl`[2], or their plural forms such as `sorts`, respectively. The following descriptions illustrate an example of rewrite theory.

[2]We also use `eq` and `rl` for conditional axioms instead of `ceq` and `crl` used in the literature for simplicity.

```
sorts S1, S2 .
vars x, y : S1 .
op c : -> S1 .
op f : S1 -> S2 .
op g : S1 S2 -> S2 .
eq g(c, f(c)) = f(c) .
rl g(x, f(y)) -> f(y) if f(x) = f(c) .
```

Then we explain the relationship between the state machine model and rewriting logic below as the basis of equational abstraction.

In fact, a rewriting relation of rewriting logic does not exactly correspond to a transition of a state machine model because of the Transitivity inference rule allowing compositions of rewriting relations. Therefore we need a precise counterpart concept for a transition in rewriting logic. We define a one-step rewriting relation $t \rightarrow^1 t'$ as a rewriting relation limited to one of the following cases.

- The last inference rule used in deriving the relation is either Equality or Replacement. The condition part of Replacement may contain any rewriting relations.
- The last inference rule used in deriving the relation is Congruence where only one assumption $t_i \rightarrow t_i'$ is one-step and all the others consist of identical terms ($t_j = t_j'$ for any $j \neq i$).

For a sort s, we write $\rightarrow^1 \cap(\mathrm{Term}(s) \times \mathrm{Term}(s))$ as $\rightarrow^1_s$.

We can create a state machine from a rewrite theory as follows. Let $\mathcal{R}$ be a rewrite theory including the following information.

- The following theory `BOOL`:

  ```
  sort Bool .
  ops true, false : -> Bool .
  op not_ : Bool -> Bool .
  op _and_ : Bool Bool -> Bool .
  ```

  as well as the equations about boolean algebras. For example,

  ```
  var X : Bool .
  eq not(false) = true .
  eq X and false = false .
  ```

- The specifications of the initial states and the labeling operator:

  ```
  sorts State, AP .
  op init : State -> Bool .
  op _|=_ : State AP -> Bool .
  ```

  where `State` and `_|=_` can be replaced with any sort and any similar operator respectively. For a term representing a state s, `init(`s`) = true` means that s represents an initial state.

In addition, we assume that there are only a finite number of closed terms for the sorts `State` and `AP`. Then the state machine $\mathcal{K}(\mathcal{R}, \texttt{State}, \texttt{\_|=\_})$ is defined as $(\mathrm{Term}(\texttt{State})/\texttt{=}, S_0, (\rightarrow^1_{\texttt{State}})^\bullet, L_{\texttt{\_|=\_}})$, where:

- $S_0 = \{[s]$ (the equivalence class to which the term s belongs) $|s$: closed term of the sort `State` and `init(`s`) = true`$\}$.
- $L_{\texttt{\_|=\_}}([s]) = \{p$: closed term of the sort `AP` | `(` s `|=` p `) = true`$\}$.

3. ASPECT MODEL IN REWRITING LOGIC

We define the model of aspect-oriented behavioral specifications in rewriting logic in the following direction.

- We first define a *behavioral specification rewrite theory (BSRT)* to model behavioral specifications in general. A BSRT treats the behaviors as evolutions of terms called *configurations* by the rewriting relations. A configuration consists of an environment and a continuation. An *environment* is an assignment of values to the variables of the specifications (not the variable symbols of the rewrite theory) at each moment. A *continuation* is an expression of a behavioral specification representing the behaviors to be executed just after the moment.

  For example, suppose `(x : 0)` and `(x = 1)` are terms representing an environment and a continuation respectively. The former term means that the value `0` is assigned to the variable `x`. Note that `x` is treated as a constant symbol in rewriting logic as well as `0` and `1`. The latter denotes the behavior that stores the value `1` to `x` and terminates. Then the configuration `config((x : 0), (x = 1))` is rewritten to `config((x : 1), end)` in which `end` denotes the termination of the behaviors.

- We then define an *aspectual rewrite theory (ART)* as a BSRT specifying the base system and the aspects. The behavioral specifications of the base system and the advices are separately defined in one ART.
- An ART consists of terms called *augmented configurations*. An augmented configuration represents a state of the entire system in which the aspects are woven into the base system. Each augmented configuration consists of the following three elements: (1) the current system state, (2) the current continuation of the entire system, and (3) the data indicating either the aspect whose advice is currently executed or the fact that the base system is currently executed. The part of the current continuation of the entire system may be a behavioral specification of the base system or one of the advices.
- The actual specifications of the entire system in which the aspects are woven are given by an extension of the ART, called *augmented ART*. The extended part consists of rewrite rules. Some of them specify the behaviors at the beginnings of and at the ends of the advice executions. The remaining rules are those transformed from the specifications of the base system and the advice behaviors.
- An ART defines the join points by specifying the starting point of an advice execution and the point at which the advice execution finishes and the control returns to the base system. Many aspect-oriented languages including AspectJ specifies join points by pointcuts and advice types (before, after, around, etc.).
- The system specified by an ART can have multiple aspects. However, we assume that the system can execute only one advice at a time. Thus we do not deal with aspect compositions. On the other hand, multiple aspects may be woven at the same join point. Our

model assume that the order of weaving is nondeterminisitic. This may lead to some type of the aspect interference issues [16]. We assume these restrictions to make our model as simple as possible and able to deal with the approach of [12]. As described in Section 6, it would be possible to relax this restriction so as to, for example, make the model deal with aspect compositions.

- In order to represent the dynamic join point model, an ART specifies the conditions to detect the join points by boolean-valued functions over a list of augmented configurations representing an execution trace.

We define BSRTs as follows.

DEFINITION 6. *A rewrite theory $\mathcal{R}$ satisfying the following conditions is called a BSRT.*

- $\mathcal{R}$ protects `BOOL`. The word *protects* means that $\mathcal{R}$ add no other terms and no other equality relations than those inferred only from `BOOL`.
- $\mathcal{R}$ has the following sorts.
  - `ENV` for *environments*.
  - `BEH` for behavioral specifications.
  - `CONFIG` for *configurations*. A configuration here consists of a pair of an environment and a behavioral specification representing the current continuation.
- $\mathcal{R}$ has the operator `config : ENV BEH -> CONFIG` that is the only operator producing a term of the sort `CONFIG`. In addition, $\mathcal{R}$ is `CONFIG`-encapsulated [15], meaning that `CONFIG` only appears as the codomain of a single operator (in this case, `config` as shown above) and does not appear as an argument in any operator in $\mathcal{R}$.

As the opposite direction of creating $\mathcal{K}(\mathcal{R}, \texttt{State}, \_| = \_)$ from $\mathcal{R}$, we can create a rewrite theory $\mathcal{R}(M)$ from a state machine $M = (S, S_0, \rightarrow, L)$ as follows.

- $\mathcal{R}(M)$ include `BOOL`.
- $\mathcal{R}(M)$ include the following constructs.

```
sorts MState, AP, APS, Env, BEH, CONFIG .
var S, S' : MState .
var P : AP .
op initState : MState -> Bool .
op member : AP APS -> Bool .
op lbl : MState -> APS .
op d : -> Env .
op beh : MState -> BEH .
op config : Env BEH  -> CONFIG .
op _|=_ : CONFIG AP -> Bool .
op trans : MState MState -> Bool .
eq (config(d, beh(S)) |= P) = member(P, lbl(S)) .
rl config(d, beh(S)) -> config(d, beh(S'))
  if trans(S, S') = true .
```

where

  - The newly introduced sorts represent the states of the machine (`MState`) and the sets of atomic propositions (`APS`), respectively. We use only meaningless values (usually only one constant `d`) as environments. This is because the actual states are stored in the second argument of `config`. However, we can also use a meaningful sort or even `MState` as the environment sort instead of `Env`. If we use `MState`, the environment and the behavior part of the configuration changes simultaneously like `config(S, beh(S)) -> config(S', beh(S'))`.
  - The operators represent the initial state predicate (`initState`), the membership function for `APS` (`member`), a representative dummy environment (`d`), the system configuration constructor (`config`), the labeling function (`_|=_`), and the transition relation (`trans`) respectively.
- For each state $s \in S$, each atomic proposition p, and each set of atomic propositions ps, constant symbols (operator symbols with no arguments) `op` s `: -> MState`, `op` p `: -> AP`, and `op` ps `: -> APS`, respectively, are included in $\mathcal{R}(M)$.
- For each atomic proposition p and each set of atomic propositions ps, the equation `eq member(`p, ps`) = true` or `= false` for the same left-hand side (LHS) according to the membership relation, is included in $\mathcal{R}(M)$. Note that we need only a finite number of these equations because the number of atomic propositions, and therefore the number of the sets of them, are finite.
- For each state $s \in S$ and each set of atomic propositions ps, the equation `eq lbl(`s`) =` ps is included in $\mathcal{R}(M)$, if $L(s) = ps$.
- For each transition $s \rightarrow s'$, an equation `eq trans(`s, s'`) = true .` is included in $\mathcal{R}(M)$.

It is straightforward to see that $\mathcal{R}(M)$ is a BSRT and $\mathcal{K}(\mathcal{R}(M), \texttt{CONFIG}, \texttt{\_|=\_})$ is equivalent to M.

Next, we define ARTs as a specific type of BSRTs.

DEFINITION 7. *An ART is a BSRT satisfying the following conditions.*

- An ART has the following sorts.
  - `ASP` for aspects and a constant indicating the base system.
  - `AC` for augmented system configurations. An augmented system configuration consists of a tuple of the terms of the sorts `ENV`, `BEH`, and `ASP`, respectively.
  - `LAC` for lists of augmented system configurations. In detail, $\mathcal{R}$ includes the operations `op nil : -> LAC` and `op [_|_] : AC LAC -> LAC`. A list is treated as an execution trace used to judge the point in which an aspect is woven.
  - `TRC` for encapsulated terms of the sort `LAC`.
- An ART also has the following operators.

- `base : -> ASP` is the constant indicating that the base system is currently executed when it is used in an augmented configuration.
- `isBase : ASP -> Bool` is the predicate that becomes true if and only if the argument is `base`. If this is false, the argument is treated as an actual aspect. Therefore such a term must not be equal to `base`.
- `adv : ASP -> BEH` produces the behavioral specification of the advice included in the aspect.
- `as : LAC ASP -> Bool` is a predicate that becomes true in the following two cases. In the first case, the second argument is an aspect whose advice can be started immediately after the base system execution represented by the trace that is the first argument. In the second case, the second argument is `base` and no advices should be started immediately. `as` stands for "aspect selection".
- `rtn : CONFIG -> Bool` is a predicate that becomes true when the argument is a state in which the advice execution finishes and the system returns to the base system execution (`rtn` stands for "return").
- `rstrt : LAC BEH -> Bool` becomes true if and only if the second argument represents a base system continuation after the advice execution is finished (`rstrt` stands for "restart").
- `ac : ENV BEH ASP -> AC` is the only operator producing the terms of the sort `AC`.
- `trc : LAC -> TRC` is the only operator producing the terms of the sort `TRC`.

- Any term t of the sort `CONFIG` satisfying `rtn(`t`) = true` cannot be rewritten without using the Equality inference rule. This means that the advice cannot be executed beyond the point to return to the base system.
- There are no equations and no rewrite rules of the terms of the sorts `AC` and `LAC`.

We can add the constructs of the behavioral specifications needed to the entire system in which the aspect is woven into the base systems.

DEFINITION 8. *Let $\mathcal{R}$ be an ART. We define $\mathcal{R}^+$ as a rewrite theory in which the following constructs are added to $\mathcal{R}$.*

- The following specifications.

```
var E : ENV .
vars B, B' : BEH .
var A : ASP .
rl trc(L)
    -> trc([ac(E, B', base) | L])
  if L = [ac(E, B, A) | _],
     isBase(A) = false,
     rtn(config(E, B)) = true,
     rstrt(L, B') = true .
rl trc(L)
    -> trc([ac(E, adv(A), A) | L])
  if L = [ac(E, B, base) | _],
     as(L, A) = true,
     isBase(A) = false .
```

- The following rewrite rules for each rewrite rule "`config(`e, b`) -> config(`e', b'`) if` c" (c is the sequence of conditions) in $\mathcal{R}$:

```
rl trc(L) -> trc([ac(e', b', base) | L])
    if L = [ac(e, b, base) | _],
    as(L, base) = true, c .
rl trc(L) -> trc([ac(e', b', A) | L])
    if L = [ac(e, b, A) | _],
    isBase(A) = false, rtn(config(e, b)) = false,
    c .
```

In this definition, the system behaviors are represented by the rewriting relation between execution traces encapsulated by the operator `trc`. However, because each rewriting step only adds a new term of the sort `AC` to the head of the list (representing the last of the trace), the step can be seen as a rewriting step for the augmented system configuration. The aim of dealing with the traces is the detection of the join points.

The four different types of the rewriting relations described in the above definition represent the following behavior types respectively: (1) restarting the base system behavior execution immediately after the advice execution is finished, (2) starting to execute the advice, (3) continuing the base system execution, and (4) continuing the advice execution.

The theorem below justifies our model with respect to the state machine model via the construction of the rewrite theory $\mathcal{R}(M)$ from a state machine M.

DEFINITION 9. *Suppose B be a state machine, ρ be a pointcut descriptor, both of which are over a set of atomic propositions AP_B, and A be an aspect machine over $AP \subseteq AP_B$. We define an ART $\mathcal{A}(B, \rho, A)$ by adding the needed specifications to $\mathcal{R}(B)$. The details are presented in Appendix A.*

THEOREM 10. *In addition to the assumptions of Definition 9, suppose B_0^ρ be a pointcut-ready machine for B and ρ. Then we have a pointcut-ready machine B^ρ and the augmented machine $\tilde{B}$ for B^ρ and A satisfying the following condition if we write $\tilde{\tilde{B}}$ for $\mathcal{K}(\mathcal{A}(B, \rho, A)^+, \texttt{TRC}, \_| = \_)$.*

$\forall l \in (2^{AP_B})^\omega$ $((\exists \pi_{\tilde{B}} :$ path in $\tilde{B}$ $(label(\pi_{\tilde{B}}) = l))$ if and only if $(\exists \pi_{\tilde{\tilde{B}}} :$ path in $\tilde{\tilde{B}}$ $(label(\pi_{\tilde{\tilde{B}}}) = l)))$

An outline of the proof is given in Appendix B.

4. EQUATIONAL ABSTRACTION IN ASPECT MODEL

Meseguer et al. proposed the equational abstraction approach as a model of abstraction for efficient model checking of rewriting logic specifications. This notion of abstraction is given in [6].

If we have a state machine $M = (S_M, S_0^M \rightarrow_M, L_M)$ and an equivalence relation $\equiv$ on S_M, we can create a *quotient state machine* $M/\equiv = (S_{M/\equiv}, S_0^{M/\equiv}, \rightarrow_{M/\equiv}, L_{M/\equiv})$ by the following definitions. We write $[s] \in S_M/\equiv$ as the equivalence class of $s \in S_M$.

- $S_0^{M/\equiv} = S_0^M / \equiv$
- For $s, s' \in S_M$, $[s] \rightarrow_{M/\equiv} [s']$ if and only if there exist $s_0 \in [s]$ and $s_0' \in [s']$ satisfying $s_0 \rightarrow_M s_0'$.
- $L_{M/\equiv}([s]) = \cap_{s_0 \in [s]} L_M(s)$

We say that $\equiv$ is *strict* if $s \equiv s' \in S_M$ implies $L_M(s) = L_M(s')$. The projection mapping $[\cdot] : S_M \rightarrow S_M / \equiv$ is called an *abstraction mapping* from M to $M/\equiv$. An abstraction mapping in the sense of [6] is a strict one. In this case, the satisfaction relation of some temporal logic is preserved by $M/\equiv$, if we define the satisfaction relation in the usual way presented in [6]. If $M' = (S_{M'}, \rightarrow_{M'}, L_{M'})$ is a state machine isomorphic to $M/\equiv$, that is, there is a bijection between A' and $S_M/\equiv$ preserving the transition relations and the labeling function, we also say M' is an abstract structure of M.

Then we describe the notion of equational abstraction that is a simplified version of [15].

THEOREM 11. *Let $\mathcal{R} = (S, \Sigma, V, E, R)$ be a rewrite theory including the specifications needed to create $\mathcal{K}(\mathcal{R}, \texttt{State}, \_| = \_)$. In addition, let E' be a set of conditional equations of the terms of the sort* `State` *and $\mathcal{R} \cup E' = (S, \Sigma, V, E \cup E', R)$. We define an equivalence relation $\equiv_{E'}$ on Term(*`State`*)$/ =_{\mathcal{R}}$ by*

$[t] \equiv_{\mathcal{R}} [t']$ if and only if $t = t'$ in $\mathcal{R} \cup E'$

Then, if the following conditions hold, $\mathcal{K}(\mathcal{R}, \texttt{State}, \_| = \_)/\equiv_{\mathcal{R}}$ is equivalent to $\mathcal{K}(\mathcal{R} \cup E', \texttt{State}, \_| = \_)$.

- *$\mathcal{R}$ is* `State`*-deadlock free, that is, $(\rightarrow^1_{\texttt{State}})^\bullet = \rightarrow^1_{\texttt{State}}$. In other words, there is at least one one-step rewriting starting from any term of the sort* `State`*.*
- *$\mathcal{R}$ is* `State`*-encapsulated.*
- *$\mathcal{R} \cup E'$ protects* `BOOL`*.*

Then we apply equational abstraction to our aspect model. Suppose that $\mathcal{R}$ is an ART with the sort of the atomic propositions `AP` and the satisfaction relation predicate `op _|=_ : TRC AP -> Bool`. It is clear that $\mathcal{R}^+$ and $\mathcal{R}^+ \cup E'$ is `TRC`-encapsulated. Therefore, if $\mathcal{R}^+$ is `TRC`-deadlock free, E' leads to an abstraction of $\mathcal{K}(\mathcal{R}^+, \texttt{TRC}, \_| = \_)$.

The compositionality of the equational abstraction of our aspect model is expressed by the following fact.

THEOREM 12. *$\mathcal{R}^+ \cup E'$ and $(\mathcal{R} \cup E')^+$ are the same.*

The proof is straightforward by observing that augmentation from $\mathcal{R}$ to $\mathcal{R}^+$ does not affect the equations.

Let $\equiv$ be the equivalence relation induced by E'. If we create Term(`TRC`)$/\equiv$ and $\rightarrow^1_{\texttt{TRC}}$ from $\mathcal{R}^+ \cup E'$, we can complete a state machine for the abstraction of the entire system by adding a labeling operator `_|=_`. Term(`CONFIG`)$/\equiv$ and $\rightarrow^1_{\texttt{CONFIG}}$ created from $\mathcal{R} \cup E'$ lead to a state machine including the abstract base system behaviors and the abstract advice behaviors separately. Therefore the coincidence of the two rewrite theories means the weaving and the abstraction operation are commutative. This fact represents the compositionality of abstraction in our approach.

5. RELATED WORK

There are many research efforts about formal behavior models of aspects [9, 1, 10, 20, 21, 5, 13, 12, 11, 4]. Some of them deal with some features of the approach of this paper. [13, 9, 12] treat finite state machine models of aspect-oriented systems mainly for the purpose of applying model checking. In addition, [9, 12] focus on compositional verification in which it is sufficient to verify the base system and the advice individually in order to verify the entire system. However, they only handle a single system at one time and do not consider relationships between systems including abstraction. [11] treats an aspect model based on the untyped lambda calculus. The main feature of this model is that it can model the (bi)simulation relation between two system expressions in a compositional way, that is, this relation is preserved under the weaving operation. However, as the literature admits, it is not easy to verify if two expressions have the (bi)simulation relation. Our model has limitations such as the first-order nature of algebraic specifications (while the untyped lambda calculus is higher-order) and our abstraction relation is a mapping. This enables us to create abstraction mappings easily. Although other formal models provide various viewpoints to aspect-oriented systems, they do not treat relationships between two systems either. [3] proposes an algebraic framework of feature-oriented development that may include AOSD. It also deals with stepwise refinement that can be considered as the inverse operation of abstraction. However, this paper does not discuss the formal correctness of the refinement. Recently, Braga [4] proposed an application of a formal framework called a constructive approach to modular structural operational semantics (constructive semantics) to aspect-oriented software. Although it does not deal with pointcuts depending on execution traces, it is promising to extend it with our approach to deal with traces.

There are also many researches about (semi-)automatic transformations of semiformal aspect behavior models mainly written in UML in the context of MDA (Model-Driven Architecture) [2, 23, 24, 17, 19, 18]. We can regard most of the transformations treated there as a generalization of the refinement relation in our approach. While we cannot discuss the correctness of the transformations in these approaches rigorously, they can treat practical situations with realistic scales and complexity. Therefore it is interesting to model them formally in our framework and evaluate the practical feasibility of our approach.

6. CONCLUSIONS

In this paper, we proposed a formal model of aspect-oriented systems based on rewriting logic and an approach of compositional equational abstraction for our model. Because our approach realizes a highly compositional way of establishing abstraction relation between aspect-oriented behavioral specifications, it is promising as a theoretical foundation of efficient AOSD methodologies.

In our approach, there are many limitations to be relaxed in the future. First, our model is too complicated and rather specific. The details of the configurations could be abstracted to the more general notion of states. Such abstraction would make our model much simpler. Our current model based on configurations could be obtained by refining the states back. The expressiveness of our approach is weak in comparison with the higher-order approaches. We also omitted the treatment of fair states. We are planning to treat aspect compositions by extending the aspect information in augmented configuration terms. We need to make

clear the limitations by trying to express various examples. Such trials will also enable us to evaluate the practical feasibility of our approach.

We need to apply our approach to concrete case studies to estimate how our approach can reduce the costs of reasoning about aspect-oriented systems. Such reasoning tasks include verification and model transformations.

Acknowledgments

This work was supported by the Ministry of Education, Culture, Sports, Science and Technology, Grant-in-Aid for Scientific Research (B). The authors would also like to thank the members of Ohsuga Lab., Honiden Lab., and GRACE center., especially Mr. Hiroyuki Nakagawa, Prof. Nobukazu Yoshioka, and Prof. Kenji Taguchi, for their enthusiastic discussions with us.

7. REFERENCES

[1] J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *Proc. of Reflection '01*, pages 187–209. Springer-Verlag, 2001.

[2] U. Aßmann and A. Ludwig. Aspect weaving with graph rewriting. In *Proc. of GCSE '99*, pages 24–36, London, UK, 2000. Springer-Verlag.

[3] D. Batory. Feature-oriented programming and the AHEAD tool suite. In *Proc. of ICSE 2004*, pages 702–703, Washington, DC, USA, 2004. IEEE Computer Society.

[4] C. Braga. A constructive semantics for basic aspect constructs. In *Semantics and Algebraic Specification*, pages 106–120, 2009.

[5] G. Bruns, R. Jagadeesan, A. S. A. Jeffrey, and J. Riely. muABC: A minimal aspect calculus. In *Proc. Concur*, volume 3170 of *Lecture Notes in Computer Science*, pages 209–224. Springer-Verlag, 2004.

[6] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM TOPLAS*, 16(5):1512–1542, September 1994.

[7] Edmund M. Clarke, Orna Grumberg, and A. Peled. *Model Checking*. MIT Press, 1999.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[9] M. Goldman and S. Katz. MAVEN: Modular aspect verification. In *Proc. of TACAS'07*, pages 308–322, 2007.

[10] R. Jagadeesan, A. S. A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *Proc. of ECOOP '03*, pages 415–427. Springer-Verlag, 2003.

[11] R. Jagadeesan, C. Pitcher, and J. Riely. Open bisimulation for aspects. In *Proc. of AOSD '07*, pages 107–120, New York, NY, USA, 2007. ACM.

[12] E. Katz and S. Katz. Modular verification of strongly invasive aspects. In *Languages: From Formal to Natural*, pages 128–147, 2009.

[13] S. Krishnamurthi and K. Fisler. Foundations of incremental aspect model-checking. *ACM Trans. Softw. Eng. Methodol.*, 16(2):7, 2007.

[14] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[15] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. *Theoretical Computer Science*, (403):239–264, 2008.

[16] I. Nagy, L. Bergmans, and M. Aksit. Composing aspects at shared join points. In *Proc. of Intl. Conf. on NetObjectDays*, pages 69–84. Springer, 2005.

[17] P. Sánchez, L. Fuentes, D. Stein, S. Hanenberg, and R. Unland. Aspect-oriented model weaving beyond model composition and model transformation. In *Proc. of MoDELS '08*, pages 766–781, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] D. Simmonds, R. Reddy, R. France, S. Ghosh, and A. Solberg. An aspect oriented model driven framework. In *Proc. of EDOC '05*, pages 119–130, Washington, DC, USA, 2005. IEEE Computer Society.

[19] D. Stein, S. Hanenberg, and R. Unland. Expressing different conceptual models of join point selections in aspect-oriented design. In *Proc. of AOSD '06*, pages 15–26, New York, NY, USA, 2006. ACM.

[20] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *Proc. ACM SIGPLAN International Conference on Functional Programming*, Uppsala, Sweden, 2003. ACM.

[21] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.

[22] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *TOPLAS*, 26(5):890–910, 2004.

[23] J. Whittle, A. Moreira, J. Araújo, P. Jayaraman, A. Elkhodary, and R. Rabbi. An expressive aspect composition language for UML state diagrams. In *Proc. of MoDELS '07*, pages 514–528. Springer, 2007.

[24] G. Zhang, M. Hölzl, and A. Knapp. Enhancing UML state machines with aspects. In *Proc. of MoDELS '07*, pages 529–543. Springer, 2007.

APPENDIX

A. DEFINITION OF $\mathcal{A}(B, \rho, A)$

We describe the definition again. Suppose B be a state machine, ρ be a pointcut descriptor, both of which are over a set of atomic propositions AP_B, and A be an aspect machine over $AP \subseteq AP_B$. We define an ART $\mathcal{A}(B, \rho, A)$ by adding the needed specifications to $\mathcal{R}(B)$.

- The constructs needed to an ART such as the specifications of `ASP` and `adv`.
- The following constructs corresponding to A.

```
sorts State', MSSeq .
var S, S1, S2 : State .
var S', S1', S2' : State' .
var M, M' : MSSeq .
var L : LAS .
var P : AP .
var A : ASP .
op beh' : State' -> BEH .
```

```
op lbl' : State' -> APS .
op asp : State' -> ASP .
op nilS : -> MSSeq .
op [[_|_]] : State MSSeq -> MSSeq .
op adl : State -> Bool .
op pc, pm : MSSeq -> Bool .
ops _==B_ : APS APS -> Bool .
op msSeq : LAS -> MSSeq .
op path : MSSeq -> Bool .
op _|=_ : TRC AP -> Bool .
op trans' : State' State' -> Bool .
eq isBase(asp(_)) = false .
eq adv(asp(S')) = beh'(S') .
eq msSeq([ac(_, beh'(S'), asp(_)) | L]) = nilS .
eq msSeq([ac(_, beh(S), base) | L]) = nilS
  if adl(S) = true .
eq msSeq([ac(_, beh(S), base) | L])
   = [[S | msSeq(L)]]
  if adl(S) = false .
eq path([[S]]) = initState(S) .
eq path([[S2 | M]]) = true
  if M = [[S1 | _]], trans(S1, S2) = true .
eq append([[]], M) = M .
eq append([[S | M]], M') = [[S | append(M, M')]] .
eq pc(M) = true
  if path(M') = true, pm(M') = true,
     M' = append(M, _) .
eq as([ac(_, _, base) | L], base)
   = not pc(msSeq(L)) .
eq as(L, asp(S')) =
    pc(msSeq(L)) and (lbl(S) ==B lbl'(S')) .
eq rstrt([ac(_, beh'(S'), _) | _], S)
    = (lbl(S) ==B lbl'(S')) .
eq trc([ac(_, beh(S), base) |_] |= P)
   = member(P, lbl(S)) .
eq trc([ac(_, beh'(S'), A) |_] |= P)
   = member(P, lbl'(S'))
  if isBase(A) = false .
rl config(d, beh'(S1')) -> config(d, beh(S2'))
  if trans(S1', S2') = true .
```

- For each state s of A, a constant symbol `op` s `: -> State'`.
- For each transition $s \to t$ of A, an equation `eq trans'(`s, t`) = true .`
- We provide additional states and transitions as follows to cope with the case of *strongly invasive aspects* in [12]. An aspect is said to be strongly invasive if it resumes to an unreachable state of the pointcut-ready machine.
  - For each label $P \subseteq AP$, a constant `op` s_{AP} `: -> State` representing an additional state and the specification of its label `eq lbl(`s_{AP}`) =` P.
  - For each additional state constant s_{AP} and a constant t of the sort `State`, an equation representing a transition between them `eq trans(`s_{AP}, t`) = true .`
- For each constant s of the sort `State`, an equation `eq adl(`s`) = true .` or `= false .` if s is an additional one or a state of B, respectively.
- The equations specifying the semantics of `pm` that satisfies `pm`(m) = `true` if and only if the pointcut descriptor ρ matches the inverse of the machine state sequence m[3].
- For each state s and each set of elements of AP ps, `eq lbl'(`s`) =` ps `.` if and only if $L'(s) = ps$. Note that ps can be represented by a constant of the sort `APS` with respect to AP_B because $AP \subseteq AP_B$.
- For each pair of constant symbols of the sort APS representing a set of atomic propositions $ps_1 \subseteq AP_B$ and $ps_2 \subseteq AP$, the equation `eq (`ps_1 `==B` ps_2`) = true .` or `= false .` according to if $ps_1 \cap AP = ps_2$ or not.
- For each state s of the aspect machine A, the equation `eq rtn(config(d, beh'(`s`))) = true .` or `= false .` according to if s is a return state ($s \in R_A$) or not.

Some constructs are explained in detail as follows.

- Because the initial state from which the aspect machine execution starts varies according to the last state of the base machine, we specify an aspect term other than `base` by encapsulating an aspect machine state representing the initial state with the operator `asp`. Thus we also specify `eq isBase(asp(_)) = false .` The advice extraction operator `adv` takes out the encapsulated state by `eq adv(asp(S')) = S' .`
- Although [12] treats only one aspect machine at a time, we can see that the framework of the literature implicitly deals with multiple aspects by the above observation. In addition, [12] specifies transitions between states of the base system and the advice that have the same label (the set of atomic propositions satisfied at a state). We can model these situations by adding the following constructs to the rewrite theory.
  - The sort `MSSeq` of sequences of the base machine states. Terms of this sort is used to detect the join points by checking them with the pointcut descriptor ρ. Such terms are extracted from the execution traces. Accordingly, we add the specification of the extraction operator `msSeq` and the pointcut predicate `pc : MSSeq -> Bool` that produces true if and only if the pointcut descriptor matches the state sequence.
  - Specifications of the equality operator `_==_` on the sort `APS` and `as`.
- For a machine state sequence m, `pc`(m) = true if and only if the inverse of m is a latter part $s_{i+1} \dots s_n$ of a path $\pi = s_0 s_1 \dots s_n$, includes only the states of B, s_i is not a state of B (that is, a state of A or an additional state), and is matched by ρ. As we show in the proof of the main theorem, we can consider the last state of the path as a pointcut state.
- `path` produces true if and only if its argument represents a finite path of B starting form an initial state.

B. PROOF OUTLINE OF THEOREM 10

First we construct B^ρ from B_0^ρ by adding the following constructs.

- For each label $P \subseteq AP$, an additional state s' satisfying $L_{B^\rho}(s') = P$.

[3]We assume that such algebraic specifications of ρ exists

- For each additional state s' introduced above and each state s of B^ρ, a transition $s' \to s$. The latter state may be an additional one or one originally in B^ρ_0.

Because all the additional states are unreachable in B^ρ, it is easy to see that B^ρ is also a pointcut-ready machine for B and ρ.

Next the transitions of $\tilde{\tilde{B}}$ are classified by the following lemma.

DEFINITION 13. *We write $e, s, a, l \rightrightarrows e', s', a', l'$ for a one-step rewriting relation in an augmented ART $\mathcal{R}^+$* `trc([ac(e, beh(s), a) | l] →`
`trc([ac(e', beh(s'), a') | l'],` *or* `beh'` *instead of* `beh` *if its argument s or s' is a state of A.*

LEMMA 14. *A transition of $\tilde{\tilde{B}}$ is obtained from either one of the following four types of one-step rewriting relations.*

1. $\mathtt{d}, s, \mathtt{asp}(s'), l \rightrightarrows \mathtt{d}, t, \mathtt{base}, l'$ *for a return state s of A and a state t of B where $L_A(s) = L_B(t) \cap AP$ or an additional state t where $L_A(s) = L_B(t)$.*
2. $\mathtt{d}, s, \mathtt{base}, l \rightrightarrows \mathtt{d}, t, \mathtt{asp}(t), l'$ *for a state s of B and an initial state t of A where $L_B(s) \cap AP = L_A(t)$ and* `pc(msSeq([[s|l]])) = true`.
3. $\mathtt{d}, s, \mathtt{base}, l \rightrightarrows \mathtt{d}, t, \mathtt{base}, l'$ *for two states s and t of B where not* `pc(msSeq([[s|l]])) = true`.
4. $\mathtt{d}, s, \mathtt{base}, l \rightrightarrows \mathtt{d}, t, \mathtt{base}, l'$ *for an additional state s and a constant t of the sort* `State`.
5. $\mathtt{d}, s, \mathtt{asp}(s'), l \rightrightarrows \mathtt{d}, t, \mathtt{asp}(s'), l'$ *for two states s and t of A where s is not a return state.*

Proof: This lemma can be proven by the fact that a one-step rewriting relation of $\mathcal{A}(M, \rho, A)^+$ can be obtained only by applying either one type of the rewrite rule in Definition 8 and each corresponding pair of conditions are equivalent. □

Fix an $l \in (2^{AP_B})^\omega$. Because the proofs of the two directions of "if and only if" are almost symmetric, we show only the "only if" part below. Thus we also fix a path $\pi_{\tilde{B}}$ of $\tilde{B}$ where $\mathrm{label}(\pi_{\tilde{B}}) = l$ and try to create a path $\pi_{\tilde{\tilde{B}}}$ of $\tilde{\tilde{B}}$ with the same label.

LEMMA 15. *Let $\pi_{\tilde{B}}$ be a path satisfying the left-hand side. Then we can decompose this path into the fragments $\pi^0, \pi^1, \ldots (, \pi^n)$ satisfying either one of the following two conditions, where each fragment is a finite path except the last one π^n if it exists.*

1. *π^i starts from an state s of B^ρ ($s \in S_0^{B^\rho}$), ends with a pointcut state s' of B^ρ, that is, a state satisfying pointcut$\in L_{B^\rho}(s')$, if π^i is finite. Every other state is in S_{B^ρ} and not a pointcut state.*
2. *π^i starts from an initial state s of A ($s \in S_0^A$), ends with a return state s' of A, if π^i is finite. Every other state is in A and not a return state.*

Proof: It is straightforward to define π^i's by induction on i by taking the longest fragments satisfying the two conditions alternately. □

Let $l \in (2^{AP_B})^\omega$ be a label, $\pi_{\tilde{B}}$ be a path of $\tilde{B}$ satisfying $\mathrm{label}(\pi_{\tilde{B}}) = l$, and $\pi^0, \pi^1, \ldots (, \pi^n)$ be a decomposition of $\pi_{\tilde{B}}$ as in Lemma 15. We can create a path $\pi_{\tilde{\tilde{B}}}$ of $\tilde{\tilde{B}}$ by composing the fragments $\pi'^0, \pi'^1, \ldots (, \pi'^n)$ shown in the following lemma.

LEMMA 16. *Under the above assumptions, there is a sequence of path fragments $\pi'^0, \pi'^1, \ldots (, \pi'^n)$, where each fragment is finite except the last one π'^n if it exists, satisfying $\mathit{label}(\pi^i) = \mathit{label}(\pi'^i)$ for each i, and the following two conditions.*

1. *If π^i satisfies the conditions 1 of Lemma 15, π'^i starts from* `d`, *s,* `base`, *l, where s is a state of B or an additional state. If π^i is finite, it also ends with* `d`, *s',* `base`, *l', where* `pc(msSeq(l')) = true`. *In addition, every other state is in S_B or an additional state constant.*
2. *If $\pi^i = s_0 s_1 \ldots (s_n)$ (s_n exists only if π^i is finite-length) satisfies the conditions 2 of Lemma 15, π'^i is* `d`, *s_0,* `asp`*(s_0), $l_0 \rightrightarrows$* `d`, *s_1,* `asp`*(s_0), $l_1 \rightrightarrows$* `d`, *s_2,* `asp`*(s_0), $l_2 \rightrightarrows \ldots (\rightrightarrows$* `d`, *$s_n$,* `asp`*$(s_0)$, $l_n)$.*

Proof: We can prove this by induction on i.

- If $\pi^0 = \pi$, this is an infinite path of B^ρ. By the definition of B^ρ, there is an infinite path of B with the same label as π. It is easy to obtain the desirable π'^0.
- If $\pi^0 = s_0 s_1 \ldots s_{i_0}$ is finite, it can be extended to an infinite path π_0^* because of the totality of $\to_{B^\rho}$. Then there is an infinite path of B^ρ with the same label as π_0^*. It is easy to obtain the desirable π'^0 by limiting the length of this path.
- To examine the cases for general i, we divide the following three cases: (1) π^i is infinite, (2) π^i is finite-length and satisfies the condition 1 of Lemma 15, and (3) π^i is finite-length and satisfies the condition 2 of Lemma 15. In case (1), we need not to proceed any more. In case (2), we can obtain π'^{i+1} by finding the next state of π'^i confirming the all the conditions of the case 2 of Lemma 14 and extending the path from it by connecting the transitions of the case 5 of the Lemma 14. In case (3), we need to be careful because π^{i+1} may start from an unreachable state. We explain this part of the proof in detail. Note that the last state s of π^{i+1} is reachable because it is a pointcut state by Lemma 15 and therefore there is a path that ends with s and is matched by ρ. Let s_0 be the first reachable state of π^{i+1} and π^+ be the path composed by a path from an initial state to s_0 and the latter part of π^{i+1} starting from s_0. By the totality of $\to_{B^\rho}$, we can extend π^+ to an infinite path π^{++} by adding transitions after s. By the definition of B^ρ, we have an infinite path π'^{++} of B such that $\mathrm{label}(\pi^{++}) = \mathrm{label}(\pi'^{++})$. Let the state of π'^{++} corresponding to s_0 and s be s'_0 and s, respectively, and $s'_0 s'_1 \ldots s$ be the subsequence of π'^{++}. If we also let $s'_{-k} s'_{-k+1} \ldots s'_{-1}$ be the sequence of additional state constant of $\mathcal{A}(B, \rho, A)$ that has the same label as $s_{-k} s_{-k+1} \ldots s_{-1}$ that is the initial segment of π^{i+1} before s_0, we have the following path of $\tilde{\tilde{B}}$: `d`, s'_{-k}, `base`, $l_0 \rightrightarrows$ `d`, s'_{-k+1}, `base`, $l_1 \rightrightarrows \ldots$ `d`, s'_{-1}, `base`, $l_{k-1} \rightrightarrows$ `d`, s'_0, `base`, $l_k \rightrightarrows$ `d`, s'_1, `base`, $l_{k+1} \rightrightarrows \ldots \rightrightarrows$ `d`, s', `base`, l, where the initial part until s'_0 consists of the additional transitions. It is straightforward to see that the execution trace makes the `pc` operator true because $\mathrm{label}(\pi^{++}) = \mathrm{label}(\pi'^{++})$ implies that ρ matches the both paths and the latter part of the trace is also a latter part of π'^{++}. □