PAPER

A Labeled Transition Model A-LTS for History-Based Aspect Weaving and Its Expressive Power

Isao YAGI[†a)], ***Nonmember***, **Yoshiaki TAKATA**[†b)], ***and*** **Hiroyuki SEKI**[†c)], ***Members***

SUMMARY This paper proposes an event-based transition system called A-LTS. An A-LTS is a simple system consisting of two agents, a basic program and a monitor. The monitor observes the behavior of the basic program and if the behavior matches some pre-defined pattern, then the monitor interrupts the execution of the basic program and possibly triggers the execution of another specific program. An A-LTS models a common feature found in recent software technologies such as Aspect-Oriented Programming (AOP), history-based access control and active database. We investigate the expressive power of A-LTS and show that it is strictly stronger than finite state machines and strictly weaker than pushdown automata (PDA). This implies that the model checking problem for A-LTS is decidable. It is also shown that the expressive power of A-LTS, linear context-free grammar and deterministic PDA are mutually incomparable. We also discuss the relationship between A-LTS and *pointcut/advice* in AOP.

key words: *labeled transition system, pushdown automaton, formal model, aspect-oriented programming, AspectJ*

1. Introduction

In this paper, we consider a simple system consisting of two agents, namely a **basic program** and a **monitor**. The monitor observes the behavior of the basic program and if the behavior matches some pre-determined pattern, such as deviation from a security policy, then the monitor interrupts the execution of the basic program and possibly triggers the execution of a specific program (e.g., an error handler). As described in detail later, this kind of systems can be found in a few fields of computer science, such as history-based access control in system security, aspect-oriented programming (AOP) and active database.

This paper proposes an event-based system called **A-LTS** for modeling the above-mentioned systems. An A-LTS is a set of finite state machines (FSMs) consisting of a basic program, a monitor and other machines that are inserted (or woven) into the basic program. The aims of the paper are two-fold: One of them is rather theoretical. Since an A-LTS is a transition system, it is natural to consider an A-LTS as a language recognizer that accepts every sequence of events that brings the A-LTS from the initial state to a final state. Thus, it is interesting to compare the expressive power of A-LTS with those of standard models such as FSM and pushdown automata (PDA). The other is to investigate automatic verification (or model checking [6]) method for systems that can be modeled as A-LTS. Among a number of types of formal verification, we are interested in formal verification in which a verification property is modeled as a subset of execution histories of a system. This type of formal verification clearly captures behavior of a system, and by appropriate modeling, automatic verification called model checking can be performed. Much research on formal modeling of this kind of systems has been done; however, a simple and clear model suitable for model checking has not been established yet (see Sect. 2.1.2). Furthermore, these two aims are closely related since there is certain relation between the language expressive power and the decidability of model checking. Thus, clarifying the expressive power of a new class of system models may shed light on the decidability and complexity of model checking of the class.

We show that the expressive power of A-LTS is strictly stronger than FSM and strictly weaker than PDA under language equivalence, bisimulation, and isomorphism. This result implies that formal verification of a program modeled as an A-LTS is decidable using a model checking method for PDA [9]. Next we compare in detail the expressive power of A-LTS with a few subclasses of PDA (or equivalently of context-free grammars): classes of deterministic PDA and linear grammars.

A-LTS resembles PA (pointcut and advice, cf. Sect. 2.1.1) of AOP languages such as AspectJ [3], but is simper than PA in the following sense:

1. An A-LTS is a data-less (or value-free) event-based transition system.
2. A basic program of an A-LTS is an FSM.

We assume a value-free system since if we allow an infinite domain, automatic verification becomes impossible. When we conduct model checking of a real-world program, we first construct an abstract model that approximates the program by making the value domain finite. As for the second point, since we are interested in the expressive power of the monitor (or pointcut), we would like to let a basic program be very simple, namely finite-state, and to know how the expressive power increases when we add the monitor.

In the rest of the paper, we will borrow the terminologies of PA in AOP since they can express many concepts on A-LTS very well although A-LTS is a very restricted model when it is considered as a formal model of PA.

Section 2 describes related work, especially studies on

Manuscript received August 3, 2006.
Manuscript revised December 1, 2006.

[†]The authors are with the Graduate School of Information Science, Nara Institute of Science and Technology, Ikoma-shi, 630–0101 Japan.
a) E-mail: isao-y@is.naist.jp
b) E-mail: y-takata@is.naist.jp
c) E-mail: seki@is.naist.jp

DOI: 10.1093/ietisy/e90–d.5.799

formal models of PA and other areas, and compares them with A-LTS. In Sect. 3, we mention the design principle of A-LTS, followed by a formal definition of A-LTS in Sect. 4. In Sect. 5, we compare the expressive power of A-LTS with FSM and PDA under language equivalence, bisimulation, and isomorphism. In Sect. 6, we state the relationship between pointcuts of A-LTS and AspectJ. Finally, we give a conclusion and future work in Sect. 7.

2. Related Work

2.1 Aspect-Oriented Programming

2.1.1 Overview

AOP is a new programming paradigm addressing the shortcomings of Object-Oriented Programming (OOP). OOP is not always suitable for describing functions and operations that cannot be encapsulated within a single class of objects (e.g., logging and synchronizing). These functions and operations are called crosscutting concerns because they straddle more than one class. AOP introduces a new module unit "aspect" for describing a crosscutting concern as a single module. In AOP, any procedure describing a crosscutting concern can be inserted into a specific execution point of a program.

Various AOP mechanisms have been proposed. Masuhara et al.[12] classify existing AOP mechanisms into the following four categories:

(1) PA in AspectJ
(2) Traversal specifications as in Demeter, DemeterJ and DJ
(3) Class composition as in Hyper/J
(4) Open classes as in AspectJ

In PA, each execution point where a procedure can be inserted is called a **join point**, and the inserted procedure is called an **advice**. When an advice is inserted into a program, we say the advice is woven into a basic program. The set of join points to which a specific advice should be connected is called a **pointcut**. An aspect is a pair of an advice and a pointcut. In (2), a concern is represented by an object called a visitor. A visitor traverses a class structure graph according to a specified traversal strategy, executing a specified advice on each visited object. A join point at which a concern is woven is thus determined statically by traversal strategy. In (3), each concern is an independent program extracted from a whole program. A join point (a location of concern in the program) is determined statically at compile time as well. In (4), a uniform change of static structure of multiple classes (e.g., method and field declaration) is regarded as a concern. Thus, a join point (a location of static structure declarations) is statically determined.

As described in Sect. 1, we are interested in formal modeling of a system whose behavior depends on execution history of a system. Hence, A-LTS is most related to PA whose dynamic behavior depends on situation in system execution. Furthermore, A-LTS can represent the recursive weaving of state machines. Recursive weaving is supported by AspectJ, and it sometimes brings unexpected behavior of a system to the system's designer. Thus it is arguable to support recursive weaving in AOP. However, in real applications such as communication protocol software, nested exceptions sometimes occur, which is a situation similar to recursive weaving. Hence, having a model that can correctly represent the recursive weaving is beneficial.

2.1.2 Formal Models of PA

There have been studies on formal modeling of PA [5], [7], [18], [20]. Douence et al.[7] proposed a formal model of PA based on a functional language Haskell. Their framework is based on the following simple principles:

- Points of interest of program execution are modeled as events.
- Each pointcut is specified as a pattern of event sequences.
- When an execution trace of a program matches a pointcut, the advice associated with the pointcut is executed.

However, the formal semantics of the pattern language, which is defined by the number of equations, is not simple, so it is not easy to use in formal verifications and other applications. Moreover, only a mechanism for selecting advice at each execution step is proposed, and one for weaving advices into a basic program is not described.

Wand et al.[20] provide denotational semantics for PA on top of an object-oriented language called BASE. A join point is an abstract runtime stack containing information on execution of procedures and advices. This enables us to define temporal pointcuts such as cflow in AspectJ in a simple way. Argument passing, also, is clearly defined by monad operations. Walker et al.[18] defines two-layered AOP, the core aspect calculus and the external language MinAML, by using simply-typed lambda-calculus. The core calculus is orthogonal to the underlying language design and a join point is an arbitrary portion of a program. In the core calculus, temporal pointcuts are defined by using regular expression to describe stack patterns. Since the main purpose of the above studies is providing formal semantics to actual AOPs, none of the studies give formal results on how the expressive power increases when PA is added to the underlying language. On the other hand, the semantics of A-LTS is value-free and finite-state, and is too simple to model PA, compared with these studies. This is partly because the purpose of this paper is to formally show how adding PA increases the expressive power as a recognizer of event sequences.

Bruns et al.[5] uses concurrency theory. In their setting, a join point is simply a message passing. They show that their language μABC can encode core MinAML. However, μABC does not consider temporal pointcuts. The temporal pointcuts in [18], [20] can be considered as a special case of the execution control known as stack inspection. As

mentioned in 2.2, the expressive power of stack inspection is known to be incomparable with that of history-based model including the monitor of A-LTS.

Nakajima et al.[14] proposed an aspect-oriented extension of UML State Diagrams. Their framework follows three principles of [7] and inherits the clarity of State Diagrams as well. Moreover, the constructs of pointcuts are simple but powerful: one can specify a pointcut such as "any configuration where a component state machine M is in a specified state s and when an event e just occurs." Switching between the basic program and a woven advice is represented by general-purpose control primitives, which can pause and resume any component state machine. Since the main purpose of [14] is model checking, the model keeps the state space of the whole model finite by prohibiting the recursion of the suspension of state machines.

2.2 Security and Database

An active database [16] consists of a set of active rules and a database instance. An active rule typically has the form of ECA (event-condition-action), which means that 'when a specified event occurs, a specified action should be performed on the current database instance if a specified condition is satisfied.' An ECA can be modeled as A-LTS by defining event-condition and action as pointcut and advice, respectively.

The access control technology most related to A-LTS is history-based access control [2], [10], [17], which is an extension of the stack inspection in Java and C♯. Schneider [17] defines an enforceable security policy as a prefix-closed nonempty set of event sequences. He also defines security automata, which exactly recognize enforceable security policies. However, the expressive power of security automaton is Turing powerful and thus too large. Fong [10] introduces several subclasses of security automata and compares their expressive power. In particular, Fong [10] defines shallow history automata with finite state space and shows that the class of policies recognized by shallow history automata is incomparable with stack inspection.

Both security automata and A-LTS take automata-theoretic approaches. The main difference between them is that in security automata, once the execution history of a controlled program deviates from a given security policy (i.e., a given pattern), the execution of the program is aborted. Hence, it is impossible for security automata including shallow history automata to simulate recursive weaving.

3. Basic Design of Program Model

A program (or a fragment of a program) is modeled as a **labeled transition system** (LTS), which defines a set of possible sequences of atomic actions (called **events**). Both a **basic program** (a program to which advices are woven) and each advice are modeled as finite LTSs, i.e., finite state machines (FSMs).

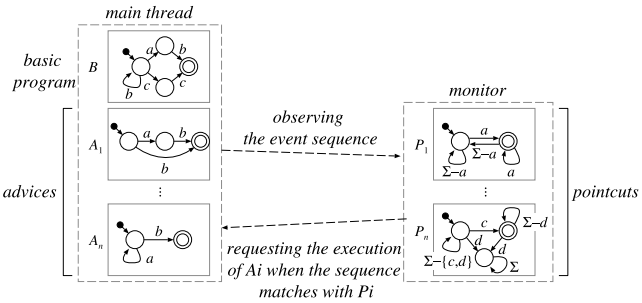


Fig. 1 A-LTS framework.

A join point where an advice is connected is determined by the following mechanism, which is similar to the one in [7]. There are two parallel synchronized virtual machines: **main thread** and **monitor** (Fig. 1). The main thread is used for the execution of a basic program and advices, and at first it invokes the basic program. The monitor observes the event sequence performed by the main thread, deciding whether the sequence matches each pointcut. When it matches pointcut P_i, the monitor tells the main thread to join advice A_i, which is associated with P_i. After the execution of A_i is finished, the main thread resumes the execution of the program that was running just before A_i was invoked.

Each pointcut is defined as a pattern of event sequences (or equivalently, a set of event sequences). When an event sequence starting at the beginning of a whole program matches a pointcut, the advice associated with the pointcut is invoked. In our model, a pointcut is defined by a deterministic finite automaton.

The execution of each advice terminates when control reaches a specified final state. Since a finite automaton can be regarded as an LTS with specified final states, we model each basic program, pointcuts, and advices as an LTS with final states.

4. Program Model A-LTS

An **A-LTS** is a tuple of a basic program, n pointcuts, and n advices. An A-LTS specifies a single infinite LTS.

4.1 Labeled Transition System

In this paper, we use LTSs with final states as the fundamental constructs. Each basic program, pointcuts, and advices is modeled as a finite LTS with final states. In a basic program and an advice, a final state is regarded as a terminating point of execution. A pointcut is used as a language acceptor. That is, for pointcut P_i, every event sequence from the initial state to a final state represents a join point specified by P_i. The behavior of a whole A-LTS is defined as an infinite LTS (in Definition 4). We only use the final states of a whole A-LTS for defining language equivalence, which shows the difference between the expressive powers of A-LTS and other models. The final states of an A-LTS can be ignored when we analyze its behavior, and thus we can use A-LTS even for modeling non-stopping systems.

Definition 1: A **labeled transition system with final states** on alphabet Σ is a 5-tuple

$$L = (\Sigma, Q_L, \rightarrow_L, I_L, F_L),$$

where Q_L is a finite or an infinite set of states, $\rightarrow_L$ ($\subseteq Q_L \times \Sigma \times Q_L$) is a transition relation, $I_L (\in Q_L)$ is an initial state, and $F_L (\subseteq Q_L)$ is a set of final states.

We denote $(q_1, a, q_2) \in \rightarrow_L$ as $q_1 \stackrel{a}{\rightarrow}_L q_2$. □

Let Q_L, $\rightarrow_L$, I_L, and F_L denote the set of states, the transition relation, the initial state, and the set of final states of an LTS L, respectively. We assume that the alphabets of all LTSs are the same and denoted by Σ.

Definition 2: An LTS L is **deterministic** if for all $q \in Q_L$ and $a \in \Sigma$, there exists exactly one $q' \in Q_L$ such that $q \stackrel{a}{\rightarrow}_L q'$. □

4.2 A-LTS

Definition 3: **A-LTS** is a $(2n + 1)$-tuple of finite LTSs

$$PR = (B, P_1, A_1, P_2, A_2, \ldots, P_n, A_n),$$

where $n \geq 0$. B is a **basic program**, $P_1, \ldots, P_n$ are **pointcuts**, and $A_1, \ldots, A_n$ are **advices**. They should satisfy the following constraints:

- Each pointcut is deterministic.
- $Q_B, Q_{A_1}, \ldots, Q_{A_n}$ are pairwise disjoint.
- The initial states are not final states for $B, P_1, \ldots, P_n, A_1, \ldots, A_n$.

□

An intuitive semantics of an A-LTS is as follows. First, the execution of B starts. When the event sequence starting at the beginning of B (time 0) matches pointcut P_i; that is, the sequence is accepted by P_i, the execution of B is suspended and advice A_i is invoked. After that, when the event sequence from time 0 grows according to the execution and matches pointcut P_j, the execution of A_i is suspended, and advice A_j is invoked. In this way, executions of advices are inserted recursively. When control reaches a final state of an advice, the suspended execution of the basic program or an advice is resumed. A-LTS terminates when control reaches a final state of the basic program.

When an event sequence simultaneously matches more than one pointcut, all advices associated with them are executed in a specific order. This order is defined by the indexes of pointcuts and advices. When a sequence simultaneously matches both P_i and P_j for $i < j$, A_i is invoked first, and A_j is invoked just after A_i terminates.

4.3 Formal Semantics of A-LTS

First we define some terminologies. The formal semantics of A-LTS is given in Definition 4. In the following, we fix an A-LTS $PR = (B, P_1, A_1, \ldots, P_n, A_n)$.

- For arbitrary set X, let X^* be the set of all finite sequences of elements in X. Let ϵ be the empty sequence. The singleton sequence that consists of element x is denoted by x itself. $\xi : \nu$ denotes the concatenation of two sequences, ξ and ν.
- Let $Q = Q_B \cup Q_{A_1} \cup \cdots \cup Q_{A_n}$. Note that by Definition 3, $Q_B, Q_{A_1}, \ldots, Q_{A_n}$ are pairwise disjoint. Let $M : Q \rightarrow \{B, A_1, \ldots, A_n\}$ be a mapping that maps $q \in Q$ to the LTS to which q belongs. For example, $\mathrm{M}(q) = B$ if $q \in Q_B$.
- A mapping $AD : Q_{P_1} \times \cdots \times Q_{P_n} \rightarrow Q^*$ is defined as follows.

  $$AD(q_1, \ldots, q_n) = I_{A_{i_1}} : I_{A_{i_2}} : \ldots : I_{A_{i_m}},$$

  where $1 \leq i_1 < i_2 < \cdots < i_m \leq n$ and $\{i_1, i_2, \ldots, i_m\} = \{i \mid q_i \in F_{P_i}\}$. Intuitively, $AD(q_1, \ldots, q_n)$ represents the list of the initial states of advices that should be started when each pointcut P_i goes to q_i. The order of the initial states in $AD(q_1, \ldots, q_n)$ corresponds to the execution order of the advices.
- A mapping $EF : Q \rightarrow Q^*$ is defined as follows.

  $$EF(q) = \begin{cases} \epsilon & \text{if } q \in F_{\mathrm{M}(q)}, \\ q & \text{otherwise.} \end{cases}$$

Definition 4: The formal semantics of an A-LTS PR is defined as the following LTS TS_{PR}.

$$TS_{PR} = (\Sigma, Q^* \times Q_{P_1} \times \cdots \times Q_{P_n}, \rightarrow_{PR}, (I_B, I_{P_1}, \ldots, I_{P_n}), F_{PR}),$$

where $F_{PR} = \{(\epsilon, q_1, \ldots, q_n) \mid q_i \in Q_{P_i} \text{ for } 1 \leq i \leq n\}$ and $\rightarrow_{PR}$ is defined by the following inference rule.

$$\frac{s \stackrel{a}{\rightarrow}_{\mathrm{M}(s)} s' \qquad q_i \stackrel{a}{\rightarrow}_{P_i} q_i' \quad (1 \leq i \leq n)}{(s : \xi,\ q_1, \ldots, q_n) \stackrel{a}{\rightarrow}_{PR} (AD(q_1', \ldots, q_n') : EF(s') : \xi,\ q_1', \ldots, q_n')}$$

□

Figure 2 shows an A-LTS PR recognizing $\{a^m b^m \mid 0 < m\}$. An A-LTS PR recognizes L if L is the set of sequences each of which brings T_{PR} to a final state (cf. Definition 8). Figure 3 is the TS_{PR} for the A-LTS PR in Fig. 2. A double circle denotes a final state. When PR in the initial configuration reads a, B and P_1 enter final states. Thus B terminates and A_1 starts. Next, A_1 can read either a or b. If A_1 reads a, then P_1 is entering a final state again, and thus A_1 is newly invoked. A_1 is recursively invoked m times just after

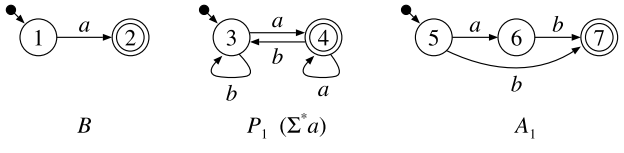


Fig. 2 A-LTS recognizing $\{a^m b^m \mid 0 < m\}$.

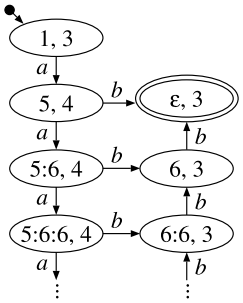


Fig. 3 TS_{PR} for PR in Fig. 2.

PR reads a^m. When a newly invoked A_1 reads b, it simply terminates. Since P_1 enters a non-final state whenever PR reads b, A_1 is not invoked at that time. Each suspended A_1 can only read b, which terminates A_1. Just after PR reads $a^m b^m$, all the suspended A_1 terminates.

5. Expressive Power of A-LTS

In this section, we compare the expressive power of A-LTS with the other state transition models, such as FSM and pushdown automaton (PDA). For two classes C_1 and C_2 of state transition models, "C_2 **includes** C_1" ($C_1 \subseteqq C_2$) if for any model M_1 in C_1, there exists some model M_2 in C_2 that is equivalent to M_1. "C_1 **is equivalent to** C_2"($C_1 = C_2$) if $C_1 \subseteqq C_2$ and $C_2 \subseteqq C_1$.

There are a few different definitions of equivalence between the two models [13]. We use three different definitions of equivalence: isomorphism, bisimulation, and language equivalence. These equivalences have the following properties:

- Two models are bisimilar if they are isomorphic.
- Two models are language equivalent if they are bisimilar.

5.1 Equivalence of Models

We define the three equivalences (isomorphism, bisimulation, and language equivalence) between two LTSs as follows.

Definition 5 (Isomorphism): LTSs L_1 and L_2 are isomorphic if there exists a bijection $\mathcal{R} : Q_{L_1} \to Q_{L_2}$ with the following properties.

(a) For any states $s_1, s_1' \in Q_{L_1}$ and any event $a \in \Sigma$, $s_1 \xrightarrow{a}_{L_1} s_1'$ if and only if $\mathcal{R}(s_1) \xrightarrow{a}_{L_2} \mathcal{R}(s_1')$.
(b) For any $s \in Q_{L_1}$, $s \in F_{L_1}$ if and only if $\mathcal{R}(s) \in F_{L_2}$. □

Definition 6 (bisimulation relation): For any pair of LTSs (L_1, L_2), a relation $\mathcal{R} \subseteqq Q_{L_1} \times Q_{L_2}$ is a **bisimulation relation** on (L_1, L_2) if for every $(s_1, s_2) \in \mathcal{R}$ and $a \in \Sigma$, $\mathcal{R}$ satisfies the following properties.

(a) If $s_1 \xrightarrow{a}_{L_1} s_1'$, then there exists some $s_2' \in Q_{L_2}$ such that $(s_1', s_2') \in \mathcal{R}$ and $s_2 \xrightarrow{a}_{L_2} s_2'$.
(b) If $s_2 \xrightarrow{a}_{L_2} s_2'$, then there exists some $s_1' \in Q_{L_1}$ such that $(s_1', s_2') \in \mathcal{R}$ and $s_1 \xrightarrow{a}_{L_1} s_1'$.
(c) $s_1 \in F_{L_1}$ if and only if $s_2 \in F_{L_2}$. □

Definition 7 (bisimulation): LTSs L_1 and L_2 are **bisimilar** if a bisimulation relation exists $\mathcal{R}$ on (L_1, L_2) such that $(I_{L_1}, I_{L_2}) \in \mathcal{R}$. □

The behavior of two models are identical if they are bisimilar. Note that the usual definition of bisimulation relation does not require property (c) of Definition 6, since the definition is over LTSs without final states. Property (c) is needed to obtain the above relation between bisimulation and language equivalence, i.e., two models are language equivalent if they are bisimilar. However, the addition of property (c) is insignificant because if we assume that LTSs L_1 and L_2 satisfy the following properties, then properties (a) and (b) imply (c) and thus Definition 6 coincides with the usual definition.

(1) At least one transition exists from every non-final state.
(2) There is no transition from final states.

Every A-LTS satisfies property (2) by Definition 4. Every A-LTS in which at least one transition exists from every non-final state of the basic program and the advices satisfies property (1). In practice this assumption does not spoil generality.

Definition 8: For an LTS L, $Lang(L) \subseteqq \Sigma^*$ is defined as follows.

$$\begin{aligned} Lang(L) = \{ & a_1 a_2 \ldots a_n \in \Sigma^* \mid \text{There exist} \\ & s_0, \ldots, s_n \in Q_L \text{ such that} \\ & I_L = s_0 \text{ and } s_{i-1} \xrightarrow{a_i}_L s_i \ (1 \le i \le n) \\ & \text{and } s_n \in F_L \}. \end{aligned}$$

$Lang(L)$ is called the **language of** L. We say that L **recognizes** a set $S \subseteqq \Sigma^*$ if and only if $S = Lang(L)$. For any sequence $w \in Lang(L)$, we say L **accepts** w. □

Definition 9 (Language equivalence): LTSs L_1 and L_2 are **language equivalent** if $Lang(L_1) = Lang(L_2)$. □

5.2 Comparisons with FSM and PDA

An FSM is an LTS with a finite number of states. The class of languages recognized by FSMs equals the class of regular languages. The class of languages recognized by pushdown automata (PDA) equals the class of context-free languages.

Now we discuss the expressive power of A-LTS. We denote the classes of A-LTSs, FSMs, and PDAs as A-LTS, FSM, and PDA, respectively. Below we will show that A-LTS $\subseteqq$ PDA and FSM $\subseteqq$ A-LTS under isomorphism and PDA $\not\subseteqq$ A-LTS and A-LTS $\not\subseteqq$ FSM under language equivalence (Theorem 2). Let $\mathcal{L}_{\text{A-LTS}}$, REG, and CFL be the classes of languages recognized by A-LTSs, FSMs, and PDAs, respectively. We will show that CFL $\not\subseteqq$ $\mathcal{L}_{\text{A-LTS}}$ and $\mathcal{L}_{\text{A-LTS}} \not\subseteqq$ REG, which imply PDA $\not\subseteqq$ A-LTS and A-LTS $\not\subseteqq$

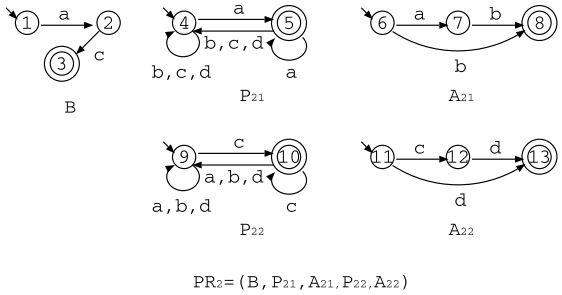


Fig. 4 A-LTS recognizing L_2.

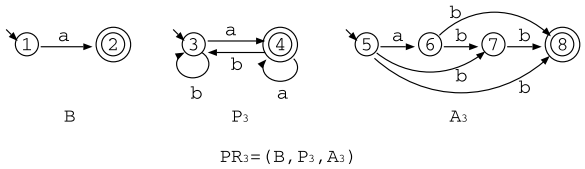


Fig. 5 A-LTS recognizing L_3.

FSM under language equivalence.

We also discuss a relation among $\mathcal{L}_{\text{A-LTS}}$ and two subclasses of CFL: the classes of deterministic context-free and linear languages. A PDA is **deterministic** if no more than one transition exists from every configuration reachable from the initial configuration. A language recognized by a deterministic PDA is a **deterministic context-free language**. A **linear context-free grammar** (or a **linear grammar**) is a context-free grammar in which at most one nonterminal symbol can occur on the right-hand side of every production. A **linear language** is a language generated by a linear grammar. Let DCFL and $\mathcal{L}_{\text{linear}}$ be the classes of deterministic context-free and linear languages, respectively.

Now we define the following eight context-free languages to discuss the inclusion relation between classes of languages.

- $L_1 = \{a^m b^m \mid 0 < m\}$
- $L_2 = \{a^m b^m c^n d^n \mid 0 < m,\ 0 < n\}$
- $L_3 = \{a^m b^n \mid 0 < m \leq n \leq 2m\}$
- $L_4 = \{a^m b^n \mid 0 \leq n \leq m,\ 0 < m\}$
- $L_5 = \{a^k b^k c^m d^n \mid 0 < m \leq n \leq 2m,\ 0 < k\}$
- $L_6 = \{a^m b^n \mid 0 < m,\ n \in \{0, m, 2m\}\}$
- $L_7 = \{a^m b^n c^k d^k \mid 0 \leq n \leq m,\ 0 < m,\ 0 \leq k\}$
- $L_8 = \{a^m b^n c^k d^k \mid n \in \{0, m, 2m\},\ 0 < m,\ 0 \leq k\}$

Lemma 1: $L_1, L_2, L_3, L_5 \in \mathcal{L}_{\text{A-LTS}}$.

[Proof] As mentioned in Sect. 4.3, the A-LTS in Fig. 2 recognizes L_1. Figures 4, 5, and 6 show A-LTSs recognizing L_2, L_3, and L_5, respectively. A-LTS PR_2 in Fig. 4 is obtained from PR in Fig. 2 by adding P_{22} and A_{22}, which resemble P_{21} and A_{21} and guarantee that the numbers of cs and ds are identical. PR_3 in Fig. 5 is obtained from PR in Fig. 2 by replacing advice A_1 with A_3, which nondeterministically consumes one or two bs for each a. A_3 thus guarantees that PR_3 exactly accepts $a^m b^n$ such that $m \leq n \leq 2m$. PR_5 in Fig. 6 is a combination of PR_2 and PR_3. □

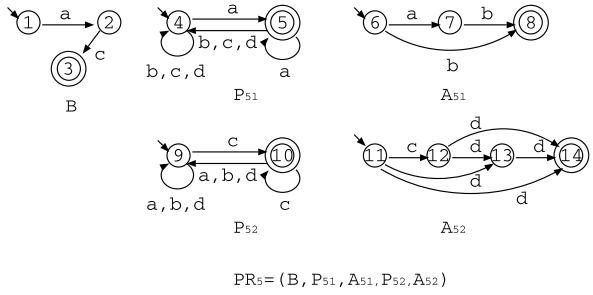


Fig. 6 A-LTS recognizing L_5.

To show that some languages are not in $\mathcal{L}_{\text{A-LTS}}$, we use the following lemma.

Lemma 2: If A-LTS PR accepts sequence w, then no pointcuts of PR accept w.

[Proof] We show the contraposition. Let w be a sequence accepted by some pointcut of PR. When the event sequence starting at time 0 becomes w, the advice corresponding to the pointcut that accepts w is invoked. Since any state precisely when an advice is invoked is not a final state of PR by Definition 4, PR does not accept w. □

Lemma 3: Let L be the language of an A-LTS PR such that $L' - \{\epsilon\} \subseteq L$ for some prefix-closed language L'. Then a constant m exists that satisfies the following condition. If $uw \in L$, $u \in L'$, $|u| \geq m$ and $w \in \Sigma^*$, then u can be decomposed into $u = xyz$ such that $|y| > 0$ and $|xy| \leq m$ and $xy^k z$ for any $k \geq 0$ also belongs to L.

[Proof] By Lemma 2, no pointcuts of PR accept any $u \in L' - \{\epsilon\}$. Therefore, since L' is prefix-closed, no advices of PR are invoked while PR reads a fixed sequence $u \in L' - \{\epsilon\}$. Let m be the cardinality of $Q_B \times Q_{P_1} \times \cdots \times Q_{P_n}$. Fix a sequence $uw \in L$ such that $u \in L'$ and $|u| \geq m$, and also fix an accepting execution of PR while reading uw. Then for the first part of the execution of PR while reading u, at least one configuration exists of PR that the execution visits twice or more, and an execution obtained by removing or repeating the part between the occurrences of the same configuration is also a valid accepting execution of PR. Letting y be the fragment corresponding to the pumped part, we obtain this lemma. □

Lemma 4: $L_4, L_6, L_7, L_8 \notin \mathcal{L}_{\text{A-LTS}}$.

[Proof] Note that each of these languages includes a^+, which equals $a^* - \{\epsilon\}$ and a^* is prefix-closed. Assuming that each of the languages is recognized by an A-LTS, then we can show a contradiction to Lemma 3 by selecting $uw = a^m b^m$ for L_4 and L_7 and $uw = a^m b^{2m}$ for L_6 and L_8 for the constant m in Lemma 3. □

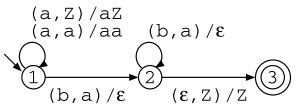


Fig. 7 Deterministic PDA recognizing L_1.

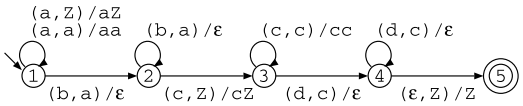


Fig. 8 Deterministic PDA recognizing L_2.

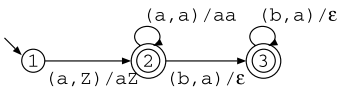


Fig. 9 Deterministic PDA recognizing L_4.

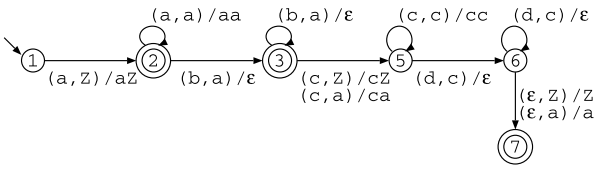


Fig. 10 Deterministic PDA recognizing L_7.

Lemma 5: $L_1, L_2, L_4, L_7 \in$ DCFL and $L_3, L_5, L_6, L_8 \notin$ DCFL.

[Proof] Figures 7, 8, 9, and 10 are deterministic PDAs that recognize L_1, L_2, L_4, and L_7, respectively. In these figures, each circle denotes a control state, and each double circle denotes a final state. The label on each transition specifies a triple $(a, g)/w$ where a is either an event or ϵ, g is a stack symbol at the top of the stack, and w is a sequence of stack symbols to which the top of the stack will be replaced [11]. Z is the start symbol of the stack. A deterministic PDA M accepts sequence w if M enters a final state just after reading w.

We prove $L_3 \notin$ DCFL by contradiction. Assume that a deterministic PDA exists that recognizes L_3. Then we can construct a PDA that recognizes the following L'_3 using a technique shown in [11, p.196].

$L'_3 = \{a^m b^n c^k \mid a^m b^n \in L_3,\ a^m b^{n+k} \in L_3\} = \{a^m b^n c^k \mid 0 < m \le n \le n + k \le 2m\}$

However, $L'_3 \notin$ CFL by the pumping lemma for context-free languages. Therefore, $L_3 \notin$ DCFL.

In a similar way, if a deterministic PDA exists that recognizes L_5, then we can construct a PDA that recognizes $L'_5 = \{a^k b^k c^m d^n e^l \mid a^k b^k c^m d^n \in L_5,\ a^k b^k c^m d^{n+l} \in L_5\} = \{a^k b^k c^m d^n e^l \mid 0 < k,\ 0 < m \le n \le n + l \le 2m\}$. Let h be a homomorphism such that $h(a) = h(b) = \epsilon$, $h(c) = a$, $h(d) = b$, and $h(e) = c$. Then $h(L'_5) = L'_3$. Since CFL is closed under homomorphism, L'_3 must be in CFL, contradicting the above fact that $L'_3 \notin$ CFL. Therefore, $L_5 \notin$ DCFL.

We prove $L_6 \notin$ DCFL by contradiction. Assume that $L_6 \in$ DCFL. Since DCFL is closed under intersection with a regular language, $L'_6 = L_6 \cap a^* b^+ = \{a^m b^n \mid n \in \{m, 2m\}, 0 < m\} \in$ DCFL. From a deterministic PDA recognizing L'_6, we can construct a nondeterministic PDA that recognizes $L''_6 = \{a^m b^n c^k \mid m > 0,\ n \in \{m, 2m\},\ n + k \in \{m, 2m\}\}$. However, $L''_6 \notin$ CFL by the pumping lemma for context-free languages. Therefore, $L_6 \notin$ DCFL.

Since $L_8 \cap a^* b^+ = L'_6$ and DCFL is closed under intersection with a regular language, $L_8 \notin$ DCFL. □

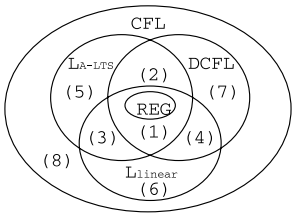


Fig. 11 Relationship between $\mathcal{L}_{\text{A-LTS}}$ and well-known classes of languages.

Lemma 6: $L_1, L_3, L_4, L_6 \in \mathcal{L}_{\text{linear}}$ and $L_2, L_5, L_7, L_8 \notin \mathcal{L}_{\text{linear}}$.

[Proof] Following linear grammars G_1, G_3, G_4, and G_6 generate L_1, L_3, L_4, and L_6, respectively.

- $G_1 = (\{S\}, \{a, b\}, \{S \to ab \mid aSb\}, S)$
- $G_3 = (\{S\}, \{a, b\}, \{S \to ab \mid abb \mid aSb \mid aSbb\}, S)$
- $G_4 = (\{S\}, \{a, b\}, \{S \to a \mid ab \mid aS \mid aSb\}, S)$
- $G_6 = (\{S, A, B, C\}, \{a, b\}, \{S \to A \mid B \mid C,\ A \to aA \mid a,\ B \to aBb \mid ab,\ C \to aCbb \mid abb\}, S)$

We can easily show that $L_2, L_5, L_7, L_8 \notin \mathcal{L}_{\text{linear}}$ by the pumping lemma for linear languages [11]. □

Lemma 7: $L_1 \notin$ REG.

[Proof] We can show this lemma by the pumping lemma for regular languages. □

Lemma 8: FSM $\subsetneqq$ A-LTS under isomorphism and REG $\subsetneqq \mathcal{L}_{\text{A-LTS}}$.

[Proof] An FSM is an A-LTS without pointcuts and advices. Therefore, FSM $\subsetneqq$ A-LTS under isomorphism, which implies REG $\subsetneqq \mathcal{L}_{\text{A-LTS}}$. □

Figure 11 shows the relationship between $\mathcal{L}_{\text{A-LTS}}$ and the classes discussed above. The following Theorem 1 states that each subset (1)–(8) in Fig. 11 is not empty.

Theorem 1: The following sets of languages are not empty.

(1) $(\mathcal{L}_{\text{A-LTS}} \cap \text{DCFL} \cap \mathcal{L}_{\text{linear}}) - \text{REG}$
(2) $(\mathcal{L}_{\text{A-LTS}} \cap \text{DCFL}) - \mathcal{L}_{\text{linear}}$
(3) $(\mathcal{L}_{\text{A-LTS}} \cap \mathcal{L}_{\text{linear}}) - \text{DCFL}$
(4) $(\text{DCFL} \cap \mathcal{L}_{\text{linear}}) - \mathcal{L}_{\text{A-LTS}}$
(5) $\mathcal{L}_{\text{A-LTS}} - (\text{DCFL} \cup \mathcal{L}_{\text{linear}})$
(6) $\text{DCFL} - (\mathcal{L}_{\text{A-LTS}} \cup \mathcal{L}_{\text{linear}})$

(7) $\mathcal{L}_{\text{linear}} - \mathcal{L}_{\text{A-LTS}} \cup \text{DCFL}$
(8) $\text{CFL} - (\mathcal{L}_{\text{linear}} \cup \mathcal{L}_{\text{A-LTS}} \cup \text{DCFL})$

[Proof] By Lemmas 1, 4, 5, 6, and 7, $L_1, L_2, \ldots, L_8$ belong to set (1), (2), ..., (8), respectively. □

The following theorem is the main result of this section.

Theorem 2: FSM $\subsetneqq$ A-LTS $\subsetneqq$ PDA under language equivalence, bisimulation, and isomorphism.

[Proof] By Lemma 8, FSM $\subseteqq$ A-LTS under isomorphism. By Theorem 1, none of the subsets (1), (2), (3), and (5) is empty. Therefore, A-LTS $\not\subseteqq$ FSM under language equivalence. For any A-LTS *PR*, there exists a PDA isomorphic to *PR*, whose set of control states is $Q_{P_1} \times \cdots \times Q_{P_n}$ and the set of stack symbols is Q. Therefore, A-LTS $\subseteqq$ PDA under isomorphism. By Theorem 1, none of the subsets (4), (6), (7), and (8) is empty. Therefore, PDA $\not\subseteqq$ A-LTS under language equivalence. □

In a general PDA, the stack is modified depending on the input event, the control state, and the topmost symbol of the stack at that time instance. In an A-LTS, however, the sequence of symbols pushed onto the stack at a transition depends on the input event and the control state (i.e., the current states of the pointcuts) and not on the contents of the stack. Moreover, the pushed sequence is uniquely determined by the event sequence starting at the beginning of the basic program since every pointcut is deterministic. These properties make A-LTS a proper subclass of PDA.

6. A-LTS and AspectJ

In this section, we discuss the relationship between the pointcuts of A-LTS and AspectJ, which is the most popular implementation of PA.

6.1 AspectJ

AspectJ is an AOP language implementing PA based on Java. A program in AspectJ consists of a set of classes and aspects. An aspect consists of pointcuts and advices. The main constructs of pointcuts are as follows.

- call(m) – the set of method calls to m.
- execute(m) – the execution of the body of method m.
- cflow(p) – the set of all join points subsequent to any join point j_p specified by pointcut p.
- get(f) – the set of execution points at which the value of data field f is used.
- set(f) – the set of execution points at which a value is assigned to data field f.

Note that for call and execute pointcuts, the execution of the whole body of method m is regarded as a single join point. For each advice, one of the three keywords {**before**, **after**, **around**} as well as a pointcut is given. Before/after denotes that the advice should be inserted before/after each join point specified by the pointcut. Around denotes that each join point specified by the pointcut should be replaced with the advice. For example, when we want to execute an advice before every method call to a method *proc*, we specify pointcut "before call(*proc*)" for the advice.

6.2 Discussion

We model a basic program and advices written in AspectJ as follows. Event set of a program is the set of join points and other related actions. For example, call to a method and assignment to a data field are regarded as events. A basic program and advices are defined as processes executing events in a specific order. As stated above, in AspectJ, the execution of the whole body of a method is regarded as a single join point. However, if we consider this join point as an atomic event, then we cannot represent the recursion of method call. Endoh et al.[8] proposed a new join point model that is finer grained than AspectJ. In the model, the start and end points of a method execution are considered as distinct join points. We follow this idea to represent "before call(m)" and "after call(m)."

Let $call_m$ be an event that represents the call to method m and $reception_m$ be an event that represents the return from m. Pointcut "before call(m)" of AspectJ is denoted as a regular language $P_{call_m} = \Sigma^* call_m$ in A-LTS. Pointcut "after call(m)" is represented as $\Sigma^* reception_m$. We consider pointcut "execute(m)" in a similar way.

Next we consider pointcut "before get(f)." Let get_f be an event that represents a reference of the value of data field f. In A-LTS, an advice is invoked just after an event that leads the current state of a pointcut to a final state. That event should not be get_f itself because we want to invoke an advice before get_f occurs. Thus, we need a prelude event $prelude_{get_f}$ that represents just before the reference of f, and we define "before get(f)" as $\Sigma^* prelude_{get_f}$. We can consider pointcut "set(f)" in a similar way.

Finally we consider pointcut "cflow(pc)." Let P_{pc} be a pointcut of A-LTS represented by a regular language that represents pc. Then "after cflow(*pointcut*)" is represented as $P_{pc} + P_{pc}\Sigma^*\Sigma_{jp}$, where Σ_{jp} is the set of all events that correspond to join points. Similarly, "before cflow(*pointcut*)" is represented as $P_{pc} + P_{pc}\Sigma^*\Sigma_{bjp}$ where Σ_{bjp} is the set of all $call_m$, all $execute_m$[†], and all the prelude events.

7. Conclusion

In this paper, we proposed a simple formal model **A-LTS** for history-based aspect weaving. We compared the expressive power of A-LTS with FSM and PDA under language equivalence, bisimulation, and isomorphism. As a result, we showed the relationship among a few subclasses of context-free languages and the class of the languages of A-LTSs,

[†]Let $execute_m$ be an event that represents the beginning of the execution of the body of method m.

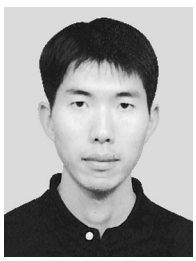
shown in Fig. 11, and FSM $\subsetneqq$ A-LTS $\subsetneqq$ PDA under language equivalence, bisimulation, and isomorphism. Finally, we stated the relationship between pointcuts in A-LTS and in AspectJ, the most popular implementation of the pointcut and advice mechanism. Since A-LTS is a subclass of PDA, formal verification of a program modeled as an A-LTS is decidable using a model checking method for PDA.

As future work, we will compare the expressive power of A-LTS with other models; e.g., context-free processes [19] and recursive state machines (RSM)[1], [4]. Moreover, we will discuss a verification of A-LTS using model checking proposed by [1], [4], because we conjecture that A-LTS is a subclass of RSM. Benedikt et al.[4] discussed the complexity of the verification of a few subclass of RSM. Following their results, we will try to find an upper and a lower bound of the complexity of the verification of A-LTS.

References

[1] R. Alur, K. Etessami, and M. Yannakakis, "Analysis of recursive state machines," 13th Conference on Computer Aided Verification (CAV 2001), LNCS 2102, pp.207–220, Paris, France, July 2001.
[2] M. Abadi and C. Fournet, "Access control based on execution history," Network & Distributed System Security Symp., pp.107–121, San Diego, USA, Feb. 2003.
[3] AspectJ Team, http://aspectj.org/
[4] M. Benedikt, P. Godefroid, and T. Reps, "Model checking of unrestricted hierarchical state machine," 28th International Colloquium on Automata, Languages and Programming (ICALP 2001), LNCS 2076, pp.652–666, Crete, Greece, July 2001.
[5] G. Bruns, R. Jagadeesan, A. Jeffrey, and J. Riely, "μABC: A minimal aspect calculus," 15th International Conference on Concurrency Theory (CONCUR 2004), LNCS 3170, pp.209–224 London, England, Aug. 2004.
[6] E.M. Clarke, Jr., O. Grumberg, and D. Peled, Model Checking, MIT Press, 2000.
[7] R. Douence, O. Motelet, and M. Sudholt, "A formal definition of crosscuts," 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (REFLECTION 2001), LNCS 2192, pp.170–186, Kyoto, Japan, Sept. 2001.
[8] Y. Endoh, H. Masuhara, and A. Yonezawa, "Continuation join point," Foundations of Aspect-Oriented Languages Workshop 2006(FOAL 2006), pp.1–10, Bonn, Germany, March 2006.
[9] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon, "Efficient algorithms for model-checking pushdown systems," 12th Conference on Computer Aided Verification (CAV 2000), LNCS 1855, pp.232–247, Chicago, USA, July 2000.
[10] P.W. Fong, "Access control by tracking shallow execution history," IEEE Security & Privacy, pp.43–55, Oakland, USA, May 2004.
[11] P. Linz, An Introduction to Formal Languages and Automata, pp.196–198, Jones and Bartlett Publishers, Sudbury, 2001.
[12] H. Masuhara and G. Kiczales, "Modeling crosscutting in aspect-oriented mechanisms," 17th European Conference on Object-Oriented Programming (ECOOP 2003), LNCS 2743, pp.2–28, Darmstadt, Germany, July 2003.
[13] M. Mukund, "From global specifications to distributed implementations," in Synthesis and Control of Discrete Event Systems, ed. B. Caillaud, P. Darondeau, L. Lavagno, and X. Xie, pp.19–35, Kluwer Academic Publishers, 2002.
[14] S. Nakajima and T. Tamai, "Aspect-oriented software design with a variant of UML/STD," 5th International Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM 2006), pp.44–50, Shanghai, China, May 2006.
[15] N. Nitta, Y. Takata, and H. Seki, "An efficient security verification method for programs with stack inspection," 8th ACM Computer & Communications Security, pp.68–77, Philadelphia, USA, Nov. 2001.
[16] N.W. Paton and O. Diaz, "Active database systems," ACM Comput. Surv., vol.31, no.1, pp.63–103, March 1999.
[17] F.B. Schneider, "Enforceable security policies," ACM Trans. on Information & System Security, vol.3, no.1, pp.30–50, Feb. 2000.
[18] D. Walker, S. Zdancewic, and J. Ligatti, "A theory of aspects," 8th ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), pp.127–139, Uppsala, Sweden, Aug. 2003.
[19] I. Walukiewicz, "Pushdown processes: Games and model checking," 8th Conference on Computer Aided Verification (CAV '96), LNCS 1102, pp.62–74, New Brunswick, July 1996.
[20] M. Wand, G. Kiczales, and C. Dutchyn, "A semantics for advice and dynamic join points in aspect-oriented programming," ACM Trans. on Programming Languages and Systems (TOPLAS), vol.3, no.5, pp.890–910, Sept. 2004.

Isao Yagi received the Ph.D. degree in information science from Nara Institute of Science and Technology, Japan, in 2006. He has been a Research Assistant Professor at Nara Institute of Science and Technology since 2006. His current research interests include formal verification of software systems.



Yoshiaki Takata received the Ph.D. degree in information and computer sciences from Osaka University, Japan, in 1997. He has been an Assistant Professor at Nara Institute of Science and Technology since 1997. His current research interests include formal specification and verification of software systems.

Hiroyuki Seki received the Ph.D. degree in information and computer sciences from Osaka University in 1987. He was with Osaka University as an Assistant Professor in 1990–1992 and an Associate Professor in 1992–1994. In 1994, he joined the faculty of Nara Institute of Science and Technology, where he has been a Professor since 1996. His current research interests include formal language theory and formal approach to software development.