JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 17, No. 2, June 2023, pp. 60-70

Using the Structure-Behavior Coalescence Method to Formalize the Action Flow Semantics of UML 2.0 Activity Diagrams

Steve W. Haga
Department of Computer Science, National Sun Yat-sen University, Kaohsiung, Taiwan, R.O.C.
stevewhaga@cse.nsysu.edu.tw

Wei-Ming Ma*
Mathematics and Physics Group of General Education Center, Air Force Academy, Kaohsiung, Taiwan, R.O.C.
k3666@gcloud.csu.edu.tw

William S. Chao
The Association of Enterprise Architects Taiwan Chapter, Kaohsiung, Taiwan, R.O.C.
architectchao@gmail.com

Abstract
The activity diagram (AD) is one of the UML 2.0 diagrams. Research has sought a precise semantic representation for the AD, partly because such representations can help to verify whether a specific AD is semantically consistent with other corresponding UML diagrams. In this study, we propose the Action Transition Graph (ATG) for semantic representation of the AD. The ATG represents the AD behavior as a finite state machine. One benefit of the ATG is that it is derived from process algebra equations, according to a precise procedure that will be formally presented. The grammar of the process algebra is also given, including an extension for representing parallel steps. This grammar allows the AD's behavior to be described by algebraic equations. Writing simple-text equations can help to simplify and structure the process of constructing ADs. In addition, these process algebra equations can be parsed by the grammar to obtain an overview diagram for ADs. The proposed overview diagram contains meaningful high-level information for the AD, and it is also shown to be directly relatable to both the underlying AD and the corresponding ATG that defines its semantic meaning.

Category: Information Retrieval / Web

Keywords: Activity diagram; Action transition graph; Structure-behavior coalescence; Process algebra

I. INTRODUCTION

The activity diagram (AD) is one of the modeling diagrams defined in the UML 2.0 [1]. The AD is a flowchart-like diagram with added features for the nested definition of activity nodes by their own ADs, and for the parallel execution of nodes between fork and join-bars.

In this study, we propose a process-algebra-based formalization of the semantics of ADs. Since algebraic equations follow a precise grammar, they can be grammatically parsed into trees. Such parse trees are equivalently representable by a diagram of nested boxes.

Open Access http://dx.doi.org/10.5626/JCSE.2023.17.2.60 http://jcse.kiise.org

This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc/4.0/) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

Received 09 March 2023; **Accepted** 02 June 2023
*Corresponding Author

Copyright © 2023. The Korean Institute of Information Scientists and Engineers pISSN: 1976-4677 eISSN: 2093-8020

We refer to this diagram of nested boxes as the AD overview diagram (AOD), because we hypothesize that it is a useful high-level representation of the AD. Moreover, the AOD derives directly from the process algebra that describes the AD, which means that the AOD has dual advantages; it is directly relatable to the AD and directly relatable to the precise semantic meaning defined by the process algebra.

The formal procedure for obtaining precise semantics is a core idea of the structure-behavior coalescence (SBC) approach [2, 3], and has been successfully applied by our team to UML sequence diagrams (SDs) [4] and UML state machine diagrams (SMDs) [5]. The SD has a visual similarity with the overview diagram proposed in [4], so the SD's representation in process algebra was straightforward. The key contribution of [4] was to use the SBC approach to obtain a labelled transition system (LTS) with the low-level formal semantics of the SD-like overview diagram. The SMD has a visual similarity with the LTS proposed in [5]—notably the proposed LTS is a valid SMD. The key contribution of [5] was therefore to use the SBC approach to provide an automatic procedure for creating an SMD-compliant LTS from a new overview diagram.

In comparison to our work in [4] and [5], UML's AD is a harder challenge, because the AD is not visually similar to either an SBC overview diagram or its derived LTS. This study presents a nontrivial example, in which the AD will be compared to its overview diagram and its semantic meaning, as represented by its LTS. Also, the procedure to automatically create an AD from either its overview diagram or its derived LTS is discussed.

Another reason why ADs are more challenging than the other UML diagrams considered in [4] and [5] is that ADs usually contain fork-join regions. Such regions describe parallel activity pathways, with the rule that actions cannot proceed beyond the join until all paths have either terminated or reached the join. This synchronization requires adding a rule to our grammar, and being novel to the field of process algebra, the precise semantic meaning of this new rule will be described in detail.

The proposed approach offers several advantages to UML ADs users. First, a process algebra representation of the AD allows a meaningful overview diagram to be created; the diagram contains important high-level information that can assist designers in understanding the general structure of a potentially complex system. Second, creating a process algebra description can ease the designing process of ADs, as the user can start with a simple text file of process algebra equations and then construct the AD from that file, using software that could be developed in accordance with the procedure in this paper. Third, the automatic construction of the AD yields a structured and consistent layout. Fourth, the proposed process algebra is formally shown, which gives a precise semantic meaning of the AD—this has been a goal of earlier researchers [5-9]. Fifth, using an LTS to represent semantic meaning can facilitate analysis by the existing LTS solvers (e.g. [10]). Moreover, our LTS representation for the AD is quite similar to the LTS representations we have previously proposed for the SD and SMD, in [4] and [5], respectively. This similarity may facilitate future work in the field of consistency checking between these diagrams.

The contents of this paper have been presented as follows. Section II discusses previous approaches for deriving the precise meaning of UML ADs. Section III presents the SBC process algebra for ADs, the resulting overview diagram (the AOD), and the corresponding semantic representation as an Action Transition Graph (the ATG). Section IV evaluates the proposed approach with a specific example. Section V offers conclusions.

II. RELATED WORK

Various representational models for defining the precise semantic meaning of UML ADs have been proposed. In most cases, the reason for defining a precise semantic meaning is to allow for consistency checking. This is also the reason why our proposed ATG representation for UML ADs is designed to closely match our previously presented LTS representations for UML SDs and SMDs.

One method that has been used for the semantic specification and validation of ADs is to convert them into equivalent abstract state machines (ASMs) [6]. An ASM specifies pseudocode to precisely describe the system behavior so that it can be validated by testing in a virtual environment. On its own, the ASM representation is not graphical, but the ASM representation is often converted into some form of the finite state machine (FSM) [6, 7]. Then, the FSM formats are compared to the format of the LTS formed by our algebraic approach.

The FSM presented in [7] uses statechart-like semantics to model ADs for workflow applications. Although limited to UML 1.0, a key feature of [7] is the use of a special FSM format that achieves parallel path execution through a path-merging mechanism that allows transitions out of states to be restricted by multiple in-edges. This merging mechanism is then used to implement the fork-join regions of UML ADs. This idea was extended by [8] to cover all the features of the UML 2.0 ADs. Although these works do create FSMs, their primary goal generally is model checking, not the idea that users would directly view the FSM. In [9], a different pseudocode algorithm is presented to create a similar form of FSM to represent all of the structures of UML ADs.

In contrast to these studies, our proposed approach builds the FSM from process algebra rather than from pseudocode. Another difference is that our proposed ATG tends to have fewer states than in the related studies because the edges in the ATG have more expressivity.

One feature that is similar between our approach and [7–9] is that our proposed FSM format (the ATG) also defines a special mechanism (parstep, described below), to implement the join node of ADs.

Defining AD behavior using compositional temporal logic (cTLA) is similar to the ASM approach [10]. TLA uses a program-like representation of system behavior via transfer protocols. Composable processes are added in cTLA. The ASM-based works, the FSM created by [10] is larger than our proposed LTS, and is not intended for direct user inspection.

Aside from FSMs, various forms of Petri nets are also often used to represent ADs. Petri nets define transitions between nodes, and the transitions can require multiple inputs to trigger. This means that Petri nets easily handle the fork-join regions of ADs. In [11], an example is provided to illustrate the intuitive way that a Petri net representation can be visually compared to the AD. Similarly, we present a visual side-by-side comparison of the AD to our representation, in Section IV. In fact, the example in [11] also presents the process algebra description of the system, although it does not use that representation in its analysis. In [12], a specific type of colored Petri net is used to represent the AD, and in [13] a specific modular Petri net is used for the representation. Although our ATG format is sufficient for analysis with an LTS solver, it is also a trivial matter to convert an LTS into a Petri net, if it were desired to perform model checking, with an existing tool for Petri nets.

A very different representation for AD semantics is to use institutions [14, 15]. Institutions represent facts about a system. By describing the behavior of each component of the AD in the syntax of institutions, the behavior can be compared for consistency checking against the behavior of the other UML diagrams that have also been represented as institutions.

One approach that is more relatable to our process-algebra-based representation is to use a similar pi-calculus representation. Despite the similarity of these representations, the related works in this area focus on solving a different problem: how to formalize the process of representing an AD with pi-calculus [16, 17]. Such a pi-calculus representation could then be used to derive our process algebra. We propose the reverse, however: not deriving the process algebra, but starting with it and deriving the AD from it.

III. STRUCTURE-BEHAVIOR COALESCENCE FOR FORMALIZING UML 2.0 ACTIVITY DIAGRAMS

This section discusses the details of the SBC method used to specify the formal semantics of the UML 2.0 ADs. This method is termed the SBC activity diagram (SBC-AD). In SBC-AD, each action flow is regarded as a process.

Table 1. Entitles of SBC-AD

Entity set	Entity name	Entity type
C	$c_1, c_2...$	Guard conditions
H	$h_1, h_2...$	Actions
Π	$\pi_1, \pi_2...$	Code snippets
R	$r_1, r_2...$	Prefixes
Ψ	$AF_1, AF_2...$	Action flows
Φ	$A_1, A_2...$	Action flow constants

A. The Entities of SBC-AD

As shown in Table 1, C refers to a set of guard conditions, H is a set of actions, Π is a set of code snippets, R is a set of prefixes, Ψ is a set of action flows and Φ is a set of action flow constants. The elements of these sets are respectively written as $c_1, c_2, \ldots ; h_1, h_2, \ldots ; \pi_1, \pi_2, \ldots ; r_1, r_2, \ldots ; AF_1, AF_2, \ldots$; and $A_1, A_2 \ldots$.

B. The Prefix Specification of SBC-AD

SBC-AD is an LTS that provides a single diagram for UML, to unify structural and behavioral constructs. In SBC-AD, each transition is labelled with a prefix that is defined as follows.

DEFINITION (PREFIX). A prefix $PX = (C, H, \Pi, R)$ consists of:

- a finite set C of optional guard conditions,
- a finite set H of actions,
- a finite set Π of optional code snippets,
- a relation $R \subseteq C \times H \times \Pi$, and $(c, h, \pi) \in R$.

The above definitions describe the labels for the ATGs edges. For any transition t_i there will be an associated relation r_i that is defined as a 3-tuple (c_i, h_i, π_i). The above definition states that the c_i and π_i fields are optional; when not needed, a *nil* is used. For a system that currently is in some state where t_i is an outgoing transition, the guard c_i will first be evaluated; if it evaluates to False, then the guard will prevent the transition from triggering -the condition of *nil* evaluates to True.

Next, if the decision is made to trigger the transition t_i, then the action h_i will be performed by the system. In special cases, if no action is to be taken then an ε action is specified (in finite state machines, epsilon transitions occur instantaneously). Finally, the code snippet π_i will be executed, if it was provided.

C. The Syntax of SBC-AD

As a formal language, SBC-AD is syntactically specified by the Backus-Naur Form (BNF) grammar shown in Fig. 1.

(1) $AF ::= ◉$

(2) $AF ::= r \bullet AF_1$

(3) $AF ::= \textbf{parstep}(PATG) \bullet AF_1$

(4) $AF ::= AF_1$ **alt** AF_2

(5) $AF ::= AF_1$ **par** AF_2

(6) $AF ::= \textbf{loop}(LATG)$

(7) $AF \overset{\textbf{ref}}{=\!=} (A)$

Fig. 1. BNF grammar of SBC-AD.

Rule (1) describes that the inactive action flow is denoted in SBC-AD by "◉".

Rule (2) describes that the action flow of executing, *in sequence*, a prefix "*r*" and then an action flow "AF_1" is denoted in SBC-AD by the expression "$r \bullet AF_1$."

Rule (3) describes that the action flow of executing, *in sequence*, a partially-defined action-transition graph (PATG) and then an action flow "AF_1" is denoted in SBC-AD by adding the designator "**parstep()**" to the sequence expression "**parstep**(*PATG*)$\bullet AF_1$". parsteps are used for modeling the fork-join regions of ADs. The format of the PATG will be discussed in Section III-F.

Rule (4) describes that the action flow of executing *an alternative* of either an action flow "AF_1" or an action flow "AF_2," is denoted in SBC-AD by the expression "AF_1 **alt** AF_2." The condition that determines whether AF_1 or AF_2 will be executed can be specified with the prefixes internal to AF_1 and AF_2. The condition can also be left unspecified, to indicate non-deterministic execution behavior.

Rule (5) describes that the action flow of executing, *in parallel*, both the action flow "AF_1" and the action flow "AF_2" are denoted in SBC-AD by the expression "AF_1 **par** AF_2."

Rule (6) describes that the action flow of executing a loop is denoted in SBC by the designator "**loop()**" in the expression "**loop(***LATG***)**." LATG is defined as an ATG that contains a cycle back to its entry node. The format of ATGs will be discussed in Section III-E.

Rule (7) describes that the action flow for executing an elsewhere-specified process "*A*" is denoted in SBC-AD by "$\overset{\textbf{ref}}{=\!=} A$." Such a process "*A*" is referred to as a *constant* or as a library process.

D. The Action/ATG Overview Diagram

A process can be described in SBC-AD by an algebraic expression conforming to the grammar presented in Fig. 1. Applying this grammar to such an expression creates a parse tree. The parse tree is equivalently represented by a diagram of nested boxes. Displaying this nested diagram can provide an intuitive visual representation of the overall behavioral structure of the process that the UML AD is describing. The nested diagram will be termed the AOD, standing for either the ATG overview diagram or the activity overview diagram.

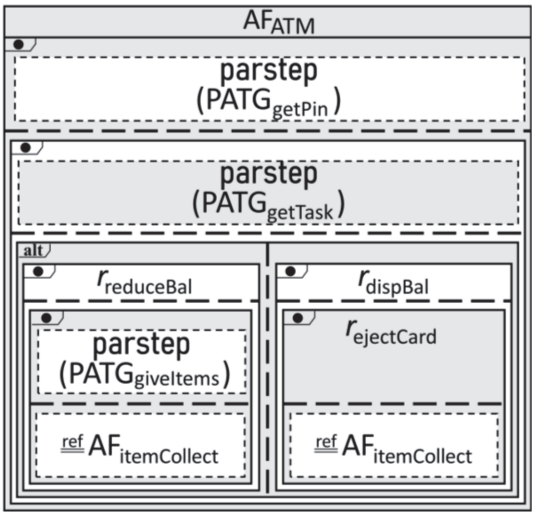


Fig. 2. An illustration of the high-level overview provided by the AOD. The semantic meaning of this example is covered in Section IV.

To show the usefulness of AODs for high-level visualization, Fig. 2 presents an example of a simple ATM that uses a variety of process algebra operators. Each nested box has a label on its upper-left corner (the operator) and a dashed line (separating the operands). The AOD is built using Fig. 1's rules to parse the algebraic expression: $AF_{ATM} \overset{\textbf{def}}{=\!=}$ **parstep(**getPin**)** • (**parstep(**getTask**)** • (($r_{reduceBal}$ • **parstep(**giveItems**)** • itemCollect) **alt** ($r_{dispBal}$ • $r_{ejectCard}$ • itemCollect))). The designer provides the "itemCollect" ATG, as well as the parstep parameters of "getPin," "getTask," and "giveItems." The semantic meaning of this ATM example will be presented in Sections IV-A and IV-B.

E. The Action Transition Graph of SBC-AD

In SBC-AD, the precise semantic meaning of a UML 2.0 AD is specified using an ATG. The ATG is a single diagram for the full system. Also, since it is based on process algebra, it is therefore a LTS [18].

DEFINITION (Action Transition Graph). An action transition graph $(ATG) = (\Psi, (\pi_0, AF_0), R, ATGR)$ consists of:

- a finite set Ψ of action flows, including an initial action flow $AF_0 \in \Psi$,
- an optional code snippet π_0 for the initial transition AF_0, where $\pi_0 \in \Pi$,
- a finite set R of prefixes,

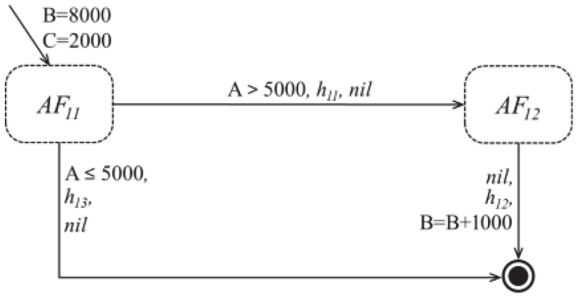


Fig. 3. The ATG for $AF_{11} \stackrel{\text{def}}{=\!=} (r_{11} \bullet r_{12} \bullet ◉)$ **alt** $(r_{13} \bullet ◉)$, where r_{11} = (A>5000, h_{11}, *nil*), r_{12} = (*nil*, h_{12}, B=B+1000), r_{13} = (A≤5000, h_{13}, *nil*), and where the code snippet of the initial transition is (B=8000; C=2000).

- a transition relation $ATGR \subseteq \Psi \times R \times \Psi$, where $(AF_i, r_j, AF_k) \in ATGR$ is written as $AF_i \xrightarrow{r_j} AF_k$.

Fig. 3 gives a sample ATG, for $AF_{11} \stackrel{\text{def}}{=\!=} (r_{11} \bullet r_{12})$ **alt** $(r_{13} \bullet ◉)$. The names in the nodes of the graph are action flows, and the labels on the edges between nodes are prefixes. There is also an initial transition into AF_{11}. Such initial transitions have a target but no source, and they are labelled with only a code snippet ("A = 8000; B = 2000"). The transition relations are $ATGR_{11} = \{(AF_{11}, (A > 5000, h_{11}, nil), AF_{12}), (AF_{11}, (A \leq 0, h_{12}, nil), ◉), (AF_{12}, (nil, h_{13}, B = B + 1000), ◉)\}$, so the action transition graph presented in Fig. 3 is expressed as $ATG_{11} = (\Psi, ((A = 8000; B = 2000), AF_{11}), R, ATGR_{11})$. In this example, the transitions from AF_{11} have mutually exclusive guard conditions, but this is not a general requirement. If the conditions for multiple transitions are all met, the choice of a trigger will be arbitrary and fair [10].

In ATGs, when $AF \xrightarrow{r_1} \cdots \xrightarrow{r_n} AF'$, then we call $(r_1 \ldots r_n, AF')$ a derivative of AF. For an initial action flow AF_0, a path $AF_0 \xrightarrow{r_1} \cdots \xrightarrow{r_n} AF_0$ would indicate a cycle back to the initial node. Thus, the ATG would meet the definition of an LATG, described by Rule (6) of Fig. 1 as $AF_0 \stackrel{\text{def}}{=\!=}$ **loop** $(LATG_0)$.

For example, Fig. 4 presents the ATG for the action

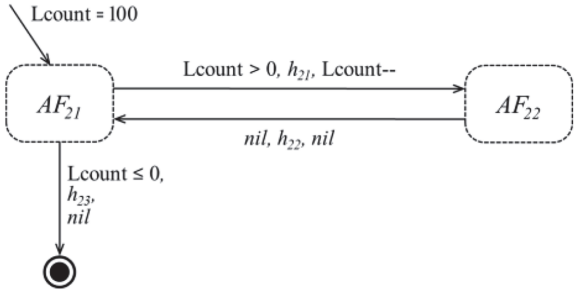


Fig. 4. A loop example defining ATG_{21}. As shown, an LATG is an ATG with a cycle to the entry node (in this case, AF_{21}). Every LATG is an ATG, so the SBC-AD representation of this example process is: $AF_{21} \stackrel{\text{def}}{=\!=}$ **loop**(ATG_{21}).

transition relation $ATGR_{21} = \{(AF_{21}, (Lcount > 0, h_{21}, Lcount--), AF_{22}), (AF_{22}, (nil, h_{22}, nil), AF_{21}), (AF_{21}, (Lcount \leq 0, h_{41}, nil), ◉)\}$, which defines $ATG_{21} = (\Psi, ((Lcount = 100), AF_{21}), R, ATGR_{21})$. The path $AF_{21} \xrightarrow{r_{21}} \cdots \xrightarrow{r_{22}} AF_{21}$ exists, so the action flow is represented as $AF_{21} \stackrel{\text{def}}{=\!=}$ **loop** $(LATG_{21})$.

F. The Transitional Semantics of SBC-AD

The semantics of SBC-AD is obtained by precise composition rules for each action flow operator. Fig. 5 gives the rules for all operators except for the parstep sequence (which is covered later).

1) The First Transition Rule for Sequence Composition

Sequence composition Rule (1) from Fig. 1 is interpreted as: for any situation, describable by $r \bullet AF_1$, the occurrence of the activity represented by prefix r infers $(r \bullet AF_1) \xrightarrow{r} AF_1$. To be precise, for any sequential action flow with a prefix that is prefixed to it, this prefix "r" is used to fulfill the transition.

Consider "AF_2", defining the action flow expression "$r \bullet AF_1$", written as "$AF_2 \stackrel{\text{def}}{=\!=} r \bullet AF_1$". The sequence composition rule for generating the code snippet of the initial transition of the action flow "AF_2" is for the initial transition of "AF_2" to acquire its code snippet from the initial transition of the action flow "AF_1." "AF_1" will not be the initial state in the new ATG, so it will no longer have its own initial code snippet.

The next sequential process can be any process. The simplest case is when it is ◉, as shown in Fig. 6. Here, it is $ATGR_{31} = \{(AF_{31}, (nil, h_{31}, nil), ◉)\}$ and $ATG_{31} = (\Psi,$

$$\textbf{Sequence} \quad \frac{}{(r \bullet AF_1) \xrightarrow{r} AF_1}$$

$$\textbf{Alternative}_1 \quad \frac{AF_1 \xrightarrow{r} AF_1'}{(AF_1 \textbf{ alt } AF_2) \xrightarrow{r} AF_1'}$$

$$\textbf{Alternative}_2 \quad \frac{AF_2 \xrightarrow{r} AF_2'}{(AF_1 \textbf{ alt } AF_2) \xrightarrow{r} AF_2'}$$

$$\textbf{Parallel}_1 \quad \frac{AF_1 \xrightarrow{r} AF_1'}{(AF_1 \textbf{ par } AF_2) \xrightarrow{r} (AF_1' \textbf{ par } AF_2)}$$

$$\textbf{Parallel}_2 \quad \frac{AF_2 \xrightarrow{r} AF_2'}{(AF_1 \textbf{ par } AF_2) \xrightarrow{r} (AF_1 \textbf{ par } AF_2')}$$

Fig. 5. The transition rules of the operators of SBC-AD. These rules provide precise semantics for the composition across the operators in an action flow algebraic expression. The transition rule for a parstep sequence is more complex, however, and is presented separately, in Fig. 10.

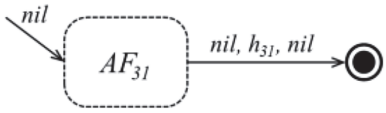


Fig. 6. An example of an ATG constructed from a sequence transition. The presented ATG_{31}, is defined as $AF_{31} \stackrel{\text{def}}{=\!=} r_{31} \bullet ◉$.

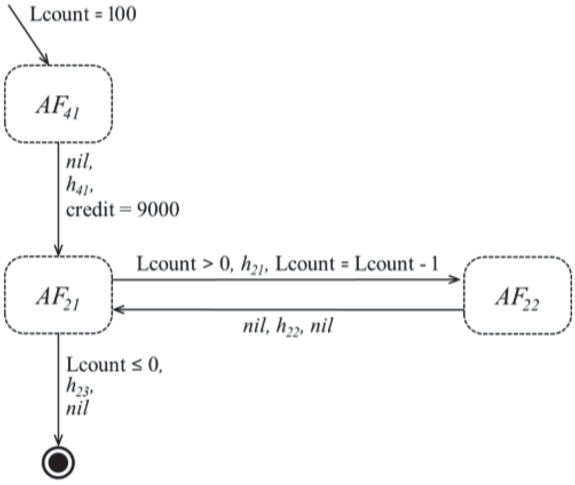


Fig. 7. An ATG constructed from a sequence transition going to a loop. ATG_{41} is defined as $AF_{41} \stackrel{\text{def}}{=\!=} r_{41} \bullet AF_{21}$, and where AF_{21} is itself defined in Fig. 4.

(nil, AF_{31}), R, $ATGR_{31}$). The case of a loop is shown in Fig. 7, where $AF_{41} = r_{41} \bullet AF_{21}$ (and where AF_{21} has been defined as a loop, in Fig. 4). Defining r_{41} as in the figure, $ATGR_{41}$ = {(AF_{41}, (nil, h_{41}, credit = 9000), AF_{21}), (AF_{21}, (Lcount > 0, h_{21}, Lcount--), AF_{22}), (AF_{22}, (nil, h_{22}, nil), AF_{21}), (AF_{21}, (Lcount ≤ 0, h_{41}, nil), ◉)} and ATG_{41} = (Ψ, ((Lcount=100), AF_{41}), R, $ATGR_{41}$).

2) The Transition Rules for Alternative Composition

The two alternative composition rules respectively, show that $AF_1 \xrightarrow{r} AF_1'$ infers (AF_1 **alt** AF_2) $\xrightarrow{r}$ AF_1' and that $AF_2 \xrightarrow{r} AF_2'$ infers (AF_1 **alt** AF_2) $\xrightarrow{r}$ AF_2'. Defining AF_3 as $AF_3 \stackrel{\text{def}}{=\!=} AF_1$ **alt** AF_2, the rule to create the code snippet of AF_3's initial transition is to concatenate the initial-transition code snippets of AF_1 and AF_2.

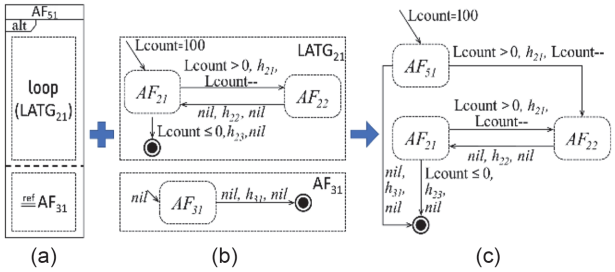


Fig. 8. An example of an ATG constructed from an alternate transition. (a) shows the AOD. (b) defines the components referenced in (a), as obtained from Figs. 4 and 6. (c) obtains the complete ATG_{51} from the transition rule of Fig. 5.

To illustrate the usage of the alternative composition rules, Fig. 8 applies them to the previously defined action flows, AF_{21} and AF_{31}, written as $AF_{51} \stackrel{\text{def}}{=\!=} AF_{21}$ **alt** AF_{31}. The obtained action transition relation is $ATGR_{51}$={(AF_{51}, (Lcount > 0, h_{21}, Lcount--), AF_{22}), (AF_{51}, (Lcount ≤ 0, h_{41}, nil), ◉), (AF_{51}, (nil, h_{31}, nil), ◉), (AF_{22}, (nil, a_{22}, nil), AF_{21}), (AF_{21}, (Lcount > 0, h_{21}, Lcount--), AF_{22}), (AF_{21}, (Lcount ≤ 0, h_{41}, nil), ◉)}, which constitutes the graph ATG_{51} = (Ψ, ((Lcount = 100), AF_{51}), R, $ATGR_{51}$).

3) The Transition Rules for Parallel Composition

The two rules for parallel composition shows, respectively, that $AF_1 \xrightarrow{r} AF_1'$ infers (AF_1 **par** AF_2) $\xrightarrow{r}$ (AF_1' **par** AF_2) and that $AF_2 \xrightarrow{r} AF_2'$ infers (AF_1 **par** AF_2) $\xrightarrow{r}$ (AF_1 **par** AF_2'). Defining $AF_4 \stackrel{\text{def}}{=\!=} AF_1$ **par** AF_2, the code snippet of AF_4's initial transition is created by concatenating AF_1 and AF_2 initial transitions' code snippets.

To illustrate the transition rules of parallel composition, Fig. 9 presents "$AF_{61} \stackrel{\text{def}}{=\!=} AF_{21}$ **par** AF_{31}."

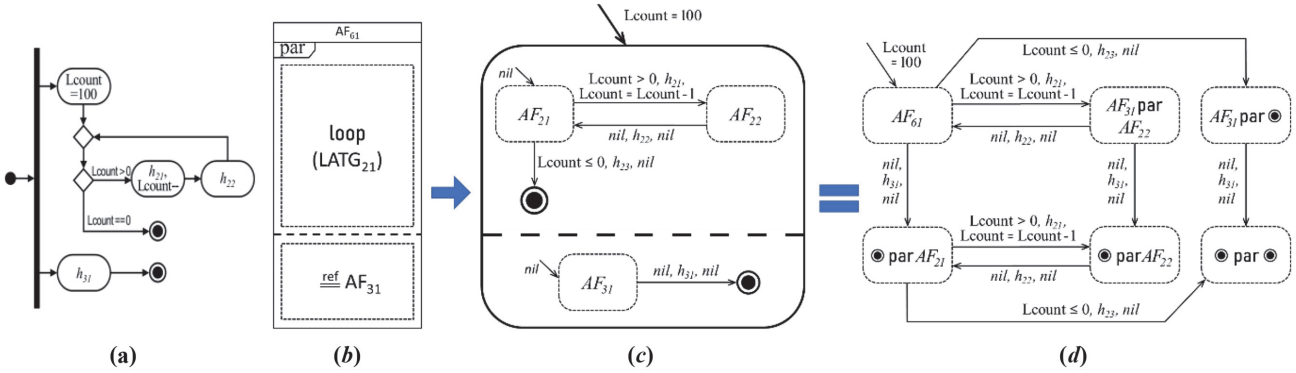


Fig. 9. An example of an ATG for a parallel transition. (a) A sample AD containing a fork, a loop, and three unspecified actions h_{21}, h_{22}, and h_{31}. (b) The corresponding AOD. (c) The corresponding orthogonal-format ATG. (d) The equivalent non-orthogonal ATG.

The action transition relation is $ATGR_{61}$={(AF_{61}, (Lcount > 0, h_{21}, Lcount--), AF_{22} **par** AF_{31}), (AF_{61}, (Lcount ≤ 0, h_{41}, *nil*), ◉ **par** AF_{31}), (AF_{61}, (*nil*, h_{31}, *nil*), AF_{21} **par** ◉), (AF_{22} **par** AF_{31}, (*nil*, h_{22}, *nil*), AF_{61}), (AF_{22} **par** AF_{31}, (*nil*, h_{31}, *nil*), AF_{22} **par** ◉), (AF_{22} **par** ◉, (*nil*, h_{22}, *nil*), AF_{21} **par** ◉), (◉ **par** AF_{31}, (*nil*, h_{31}, *nil*), ◉ **par** ◉), (AF_{21} **par** ◉, (Lcount ≤ 0, h_{41}, *nil*), ◉ **par** ◉), (AF_{21} **par** ◉, (Lcount > 0, h_{21}, Lcount--), AF_{22} **par** ◉)}, which constitutes ATG_{61} = (Ψ, ((Lcount = 100), AF_{61}), R, $ATGR_{61}$), as shown in Fig. 9(d).

The set of action flow possibilities for parallel composition is the Cartesian product of its two action flow operands. The size of this product increases significantly when both operands have many states. Hence, the orthogonal form of the finite state machine may be preferred, as shown in Fig. 9(c). The various FSMs proposed in the related works do not provide an orthogonal form, partly because those works are not aimed at allowing the users to directly view the FSM.

Yet the concept of orthogonal FSMs is natural to UML users because UML state machine diagrams also have this property. Compared to our approach, however, UML provides no explicit formalism to create a nonorthogonal FSM (e.g., Fig 9(d)) from an orthogonal one (e.g., Fig. 9(c)).

4) The Transition Rule for Parstep Sequence Composition

To illustrate parstep-sequence composition, Fig. 10 gives example "$AF_{71} \stackrel{\text{def}}{=}$ **parstep** ($PATG_{71}$) • AF_{31}". The definitions of $PATG_{71}$ and AF_{31} are given in Fig. 10(b). AF_{31} is from Fig. 6, but $PATG_{71}$ is a new example illustrating the syntax of partial ATGs. As shown, PATGs are used for fork-join regions of UML ADs, so PATGs begin with a fork-bar and end with a join-bar. Between these bars are parallel process threads (in this example: r_{71} • r_{72} • r_{73} and r_{74} • r_{75}). As shown, each thread reaches the join-bar, so the execution cannot progress further until both threads finish.

Fig. 10(c) shows the procedure for constructing the ATG for a parstep sequence transition. The procedure

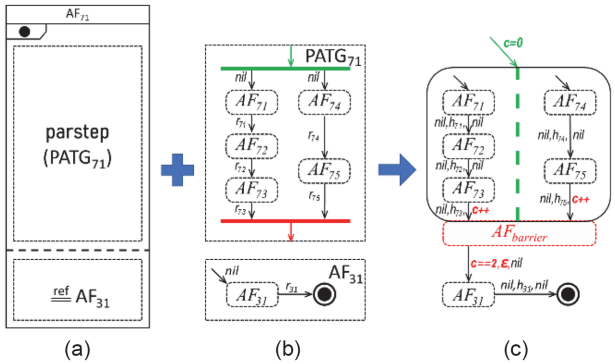


Fig. 10. Example of an ATG constructed from a parstep-sequence transition. (a) Overview diagram. (b) Definitions of the components referenced by (a). The PATG has parallel threads running between fork and join bars. (c) This example's semantic meaning. The green fork-bar of (b) has produced a green dashed orthogonal line in (c), and the red join-bar of (b) has produced a barrier state in (c), as well as code snippets (*C*++) and a guarded condition (*c*==2).

involves separately converting the PATG's fork and join-bars. To highlight these two conversions, the figure shows one in green and the other in red.

As shown, the green fork-bar of Fig. 10(b) is converted to an orthogonal format ATG. The initial code snippets of the PATG's parallel threads concatenate for the orthogonal ATG region's initial code snippet (as was also the case in the parallel composition). In this example, however, neither parallel thread of $PATG_{71}$ has a code snippet to concatenate. Nonetheless, a code snippet c=0 is seen in Fig. 10(c). Setting c to 0 indicates that no threads are finished at the time of entering the orthogonal region.

As also shown in Fig. 10(b)'s, the red join-bar is converted to a new state, $AF_{barrier}$, put at the exit of the orthogonal region. For each thread, the transition into $AF_{barrier}$ has a *C*++ code snippet, to indicate that that thread has completed. Thus, the transition out of $AF_{barrier}$ is guarded by c==2, and both of PATG's branches must finish before progressing to AF_{31}. The transition out of

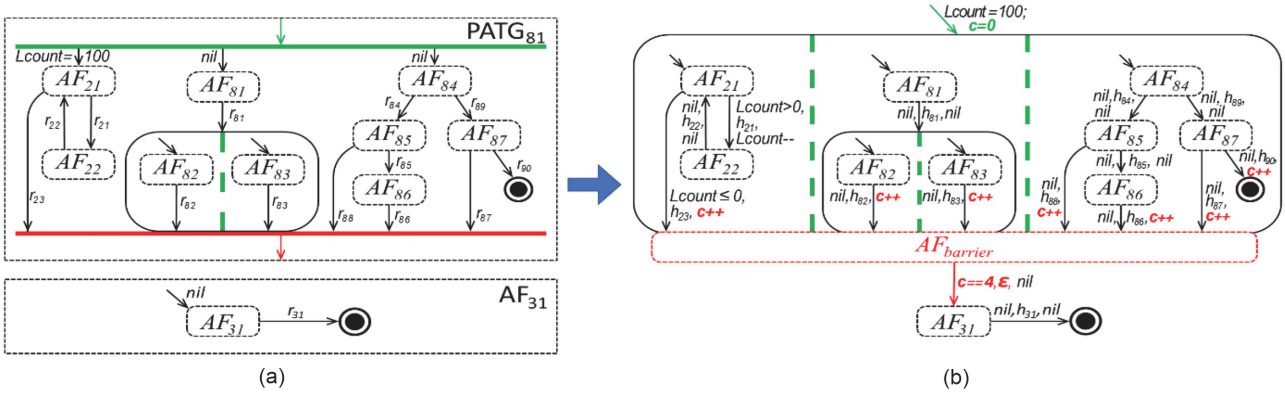
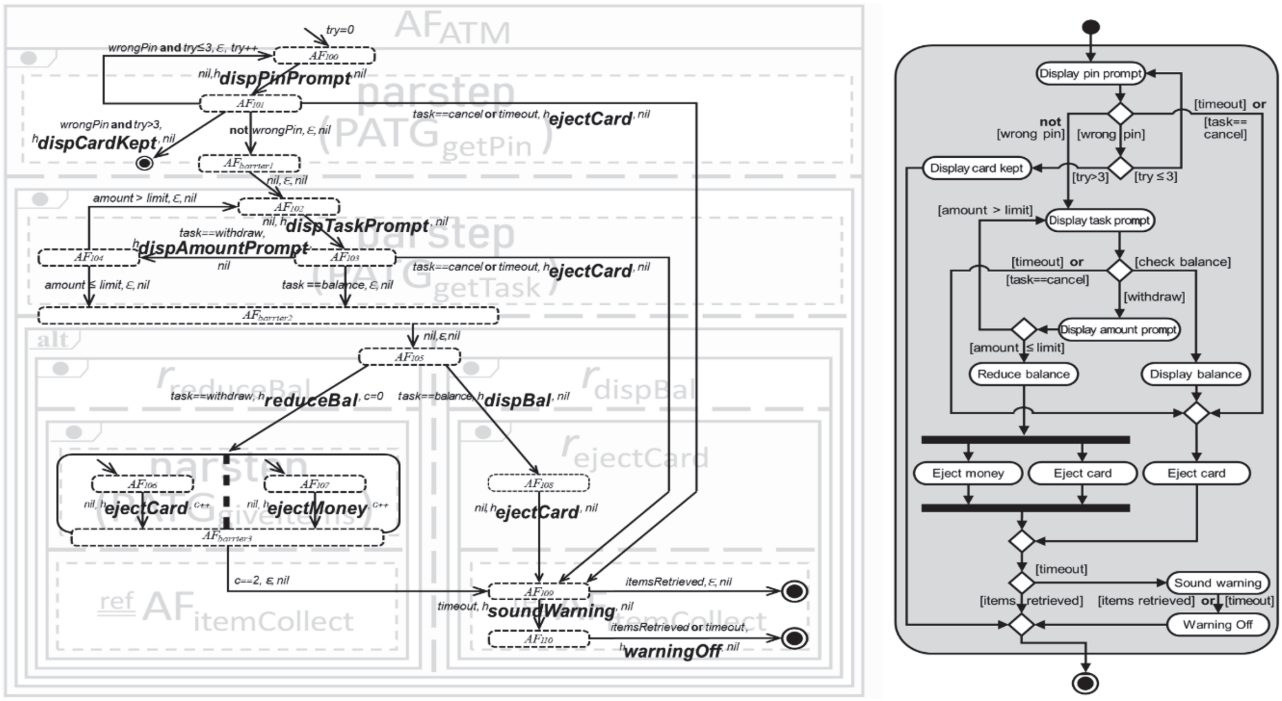


Fig. 11. A complex ATG example constructed from a parstep transition: (a) the components and (b) the corresponding orthogonal-format ATG.



(a) (b)

Fig. 12. An illustration of the correspondence between the AOD, ATG, and AD, for a simple ATM. In (a), the AOD from Fig. 2 is reproduced, with its ATG in the foreground. The ATG has been arranged so as to let each ATG region to overlay the AOD action flow producing it. (b) The corresponding AD. Note that actions occur within AD states, but on ATG transitions.

$AF_{barrier}$ also has an action of ε, which is standard state machine syntax for an instantaneous transition.

PATGs can be more complex than $PATG_{71}$; in Fig. 11, three parallel threads leave the fork. As shown in Fig. 11(a), the rightmost thread (comprising $AF_{84} - AF_{87}$) has four alternative paths ($r_{84} \bullet r_{85} \bullet r_{86} \bullet join$, $r_{84} \bullet r_{88} \bullet join$, $r_{89} \bullet r_{87} \bullet join$, and $r_{89} \bullet r_{90} \bullet ◉$). Only three reach the join-bar, but in Fig. 11(b), each path ends with *C++*, including the $r_{89} \bullet r_{90} \bullet ◉$ path. The middle thread ($AF_{81} - AF_{83}$) illustrates parallelism within a thread. Parallel paths $r_{81} \bullet r_{82} \bullet join$ and $r_{81} \bullet r_{83} \bullet join$, and both reach the join-bar. Thus, in Fig. 11(b), both have *C++* on their edges to AF_{barrier}, and both contribute to the guarded condition's bound, *c*==4. In other words, the bound is calculated as the number of swimlanes that reach the join, rather than the number that start at the fork.

Concerning the leftmost thread of Fig. 11(a), it is the LATG of Fig. 4, and shows the composition of loops in PATG threads. This capability of placing a LATG within a PATG is useful even if there is no parallelism in the PATG, because it allows for process algebra descriptions of action flows after loop exits in LATGs. Fig. 12 will give an example to clarify such a usage. When used this way, the bound on the guarded condition will be *c*==1, which means that the guard is unnecessary and can be removed along with the code snippets for *c*=0 and *C++*.

IV. EVALUATION OF THE SBC METHOD

The proposed approach offers two key benefits for AD design and validation. First, an AD overview diagram is given. This AOD corresponds directly to the AD and can simplify the process of building the AD. Further, the AOD offers meaningful high-level information and its format is similar to (and easily compared to) the UML sequence and state machine overview diagrams presented in [1] and [2]. Second, a formal procedure is given for the precise semantic meaning of the AOD (and so for the AD it represents), as an LTS. The format of this LTS is consistent with the rules of UML SMDs, so it can be used to create that diagram. To best understand these benefits, an example will now be given.

A. Comparing the AOD to the AD

Fig. 12 gives the AOD of Fig. 2, but in grey with its ATG overlaid on it (so, for a clear view, see Fig. 2). Fig. 12(b) gives the AD. All the complexity of Fig. 12(b) is summarized as: $AF_{\text{ATM}} \stackrel{\text{def}}{=} \textbf{parstep}(\text{PATG}_{\text{getPin}}) \bullet \textbf{(parstep}(\text{PATG}_{\text{getTask}}) \bullet ((r_{\text{reduceBal}} \bullet \textbf{parstep}(\text{PATG}_{\text{giveItems}}) \bullet AF_{\text{itemCollect}}) \textbf{ alt } (r_{\text{dispBal}} \bullet r_{\text{ejectCard}} \bullet AF_{\text{itemCollect}})))$, where $AF_{\text{itemCollect}} \stackrel{\text{def}}{=} ((r_{\text{itemsRetrieved}} \bullet ◉) \textbf{ alt } (r_{\text{timeout}} \bullet r_{\text{warningOff}} \bullet ◉))$.

ADs can derive from these expressions (except PATG

parts). Expression operators define the AD's shape and operand prefixes define its labels. Thus, Fig. 12(b)'s decision diamonds map to **alt** operators. Eg, the two outgoing edges from $AF_{\text{itemCollect}}$ has prefixes $r_{itemsRetrieved} = (c_{\textbf{itemsRetrieved}}, h_{dispWelcome}, nil)$ and $r_{timeout} = (c_{\textbf{timeout}}, h_{soundWarning}, nil)$, and the matching decision diamond (at the bottom of Fig. 12(b)) has edge labels ***items Retrieved*** and ***timeout***. So, the decision diamond labels derive from guard conditions in the prefixes of operands to **alt** operators. Besides edge labels, ADs have node labels that map to expression prefix actions. For example, the above $r_{timeout}$ prefix contains the action $h_{\textbf{soundWarning}}$, and the matching node of Fig. 12(b) is labeled ***Sound warning***.

As shown in Fig. 12, AODs offer useful high-level information that is easy to visually compare to ADs. Also, AODs are expressible in algebraic equations that succinctly capture the information in ADs. So, software can be used to automate constructing UML ADs from a text file of equations.

B. Comparing the AOD to the AD

Fig. 12(a) also shows the ATG overlaid on the AOD. This ATG matches to the algebraic description: AF_{ATM} **parstep(**getPin**)** ● **(parstep(**getTask**)** ● ((r$_{reduceBal}$ ● **parstep (**giveItems**)**●itemCollect) **alt** (r$_{dispBal}$ ● r$_{ejectCard}$ ● itemCollect))), where the parstep elements ($PATG_{getPin}$, $PATG_{getTask}$, and $PATG_{giveItems}$) must be supplied by the user. The ATG presented in Fig. 12(a) is a fully-defined, full-system description, as all references are replaced with their ATG representations. One feature to note in Fig. 12 is that there are four separate accesses to the $AF_{itemCollect}$ library, and those four access pathways are why state AF_{109} has four in-edges.

As AODs were meaningfully compared to ADs (Section IV-A), so too can ATGs be meaningfully compared to ADs. As can be seen in Fig. 12(a), the ATG edges are labelled with prefixes (3-tuples of a condition, an action, and a code snippet). In Fig. 12(a), there are eleven prefixes with non-ε actions, each displayed in the enlarged text; in Fig. 12(b), the AD has eleven nodes (i.e., ellipses), each labeled with text from a corresponding action in Fig. 12(a). Similarly, each nonnil guard condition in Fig. 12(a) corresponds to one of the decision-diamond edge labels in Fig.12(b)—with the two caveats. In the first caveat, the AF_{barrier3}'s c==4 guard is subsumed into the AD's join-bar, and in the second caveat the compound guard conditions (e.g., *wrongPin* **and** $try \leq 3$) have been split into separate decision diamonds in Fig. 12(b).

The most important observation to draw from Fig. 12 is that the AD can be automatically extracted (potentially by software) from the ATG, and the ATG is itself automatically extracted from the AOD. Thus, a simple set of process algebra equations (along with some PATG and/or LATG information) is enough to generate the AD. Not only does this simplify the process of creating an AD, but it also allows summarizing with an overview diagram. In addition, note that the level of detail in the AD is controlled by the number of prefixes with non-ε actions, so the user has a way to control the size and detail of the AD.

C. Comparison of SBC-AD to Other Approaches

Table 2 summarizes the capabilities of various approaches. As shown in the table, previous methods do not consider providing an overview diagram for the AD. The lack of such an overview has two consequences for all of the related works. First, these works do not assist the user either to design the AD or to understand its top-level overall behavior. Second, techniques that do not start with an overview will need added formalism to handle those ADs that may not have been designed in a hierarchical manner. In contrast, ADs that have been designed with process algebra will naturally exhibit a simple hierarchical control flow.

Table 2 also presents a rough guideline of which AD component cases are handled by various techniques. This

Table 2. A comparison of AD representations

	Top-level overview of the AD	Assists with construction of the AD	A clear procedure to represent complex AD cases involving				Formal semantics representation format
			Loop	Fork	Join	Composition	
SBC-AD	√	√	√	√	√	√	Process algebra+ATG (an FSM)
Knieke et al. [8]			√	√	√	√	Executable ASM
Raschke et al. [9]			√	√	√	√	FSM
Kaliappan et al. [10]			√	√		√	cTLA
Staines [12]			√	√			Colored Petri net
Rahim et al. [13]			√	√		√	Modular Petri net
Achouri et al. [15]				√	√		Institutions
Belghiat et al. [17]			√	√	√		Pi-calculus

checklist is only approximate. For one, a given technique might be adaptable to handle some component, but such as an adaptation was not formally discussed. For another, a technique may be able to handle simple cases for a given AD component, but not complex ones (e.g., a loop inside of a fork-join AD region).

Finally, Table 2 describes the various representation formats. Comparing the usefulness of these formats, one issue to consider is if the representation has a user-understandable, intuitive correlation to the AD, such as the one presented in Fig. 12. Another issue is whether the representation is well suited for use with analysis tools and to compare for consistency with other UML diagrams. In the case of the ATG, it can be analyzed with an LTS analyzer, and we have designed it to be very similar to the transition graphs that we have previously provided for UML sequence diagrams and state machine diagrams [4, 5].

V. SUMMARY AND CONCLUSION

A process algebra for describing UML ADs has been presented, and has been shown to naturally generate an overview diagram (AOD). The AOD provides useful high-level information about the AD. A simple formal procedure to create a precise semantic meaning of the AD is also presented. The format of the semantic representation is termed the ATG. The ATG is an FSM format that allows parallel execution paths. The ATG is comparable to our previously proposed semantic representations of UML SDs and SMDs.

In comparison to our previous works, the grammar proposed in this study introduces a parstep operator to model the fork-join regions of ADs. The precise semantic meaning of this operator is presented, so that the procedure for constructing the ATG from process algebra equations containing parsteps is fully presented. Moreover, the parstep operator is also shown to be useful in allowing partially-defined processes, even outside of fork-join regions.

Conflict of Interest(COI)

The authors have declared that no competing interests exist.

REFERENCES

1. Object Management Group, "Unified Modeling Language version 2.5," 2017 [Online]. Available: https://www.omg.org/spec/UML/2.5.1/About-UML.
2. K. P. Lin and W. S. Chao, "The structure-behavior coalescence approach for systems modeling," *IEEE Access*, vol. 7, pp. 8609-8620, 2019.
3. W. M. Ma and W. S. Chao, "Structure-behavior coalescence abstract state machine for metamodel-based language in model-driven engineering," *IEEE Systems Journal*, vol. 15, no. 3, pp. 4105-4115, 2021.
4. S. Haga, W. M. Ma, and W. S. Chao, "Structure-behavior coalescence method for formal specification of UML 2.0 sequence diagrams," *Journal of Computing Science and Engineering*, vol. 15, no. 4, pp. 148-159, 2021.
5. S. W. Haga, W. M. Ma, and W. S. Chao, "Formalizing UML 2.0 state machines using a structure-behavior coalescence method," in *Proceedings of 2022 IEEE Industrial Electronics and Applications Conference (IEACon)*, Kuala Lumpur, Malaysia, 2022, pp. 174-179.
6. S. Sarsted and W. Guttmann, "An ASM semantics of token flow in UML 2 activity diagrams," in *Perspectives of System Informatics*. Heidelberg, Germany: Springer, 2006, pp. 349-362.
7. H. Eshuis, "Semantics and verification of UML activity diagrams for workflow modelling," Ph.D. dissertation, Centre for Telematics and Information Technology (CTIT), University of Twente, Enschede, Netherlands, 2002.
8. C. Knieke and U. Goltz, "An executable semantics for UML 2 activity diagrams," in *Proceedings of the International Workshop on Formalization of Modeling Languages*, Maribor Slovenia, 2010, pp. 1-5.
9. A. Raschke, "Translation of UML 2 activity diagrams into finite state machines for model checking," in *Proceedings of 2009 35th Euromicro Conference on Software Engineering and Advanced Applications*, Patras, Greece, 2009, pp. 149-154.
10. P. S. Kaliappan and H. Konig, "On the formalization of UML activities for component-based protocol design specifications," in *SOFSEM 2012: Theory and Practice of Computer Science*. Heidelberg, Germany: Springer, 2012, pp. 479-491.
11. H. Storrle and J. H. Hausmann, "Towards a formal semantics of UML 2.0 activities," *Software Engineering*, vol. 2005, pp. 117-128, 2005.
12. T. S. Staines, "Intuitive mapping of UML 2 activity diagrams into fundamental modeling concept Petri net diagrams and colored Petri nets," in *Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, Belfast, UK, 2008, pp. 191-200.
13. M. Rahim, A. Hammad, and M. Boukala-Ioualalen, "Towards the formal verification of SysML specifications: translation of activity diagrams into modular petri nets," in *Proceedings of 2015 3rd International Conference on Applied Computing and Information Technology/2nd International Conference on Computational Science and Intelligence*, Okayama, Japan, 2015, pp. 509-516.
14. M. V. Cengarle and A. Knapp, "An institution for UML 2.0 interactions," 2008 [Online]. Available: https://mediatum.ub.tum.de/doc/1094623/1094623.pdf.
15. A. Achouri and L. J. B. Ayed, "A formal semantic for UML 2.0 activity diagram based on institution theory," *International Journal of Soft Computing and Software Engineering*, vol. 3, no. 3, pp. 199-204, 2013.
16. V. S. Lam, "On π-calculus semantics as a formal basis for UML activity diagrams," *International Journal of Software Engineering and Knowledge Engineering*, vol. 18, no. 4, pp. 541-567, 2008.

17. A. Belghiat and A. Chaoui, "A graph transformation of activity diagrams into π-calculus for verification purpose," in *Proceedings of the 3rd International Conference on Advanced Aspects of Software Engineering (ICAASE)*, Constantine, Algeria, 2018, pp. 107-114.

Steve W. Haga

Steve W. Haga received his M.E. and Ph.D. in electrical engineering from the University of Maryland, College Park, USA, in 1999 and 2005. He is currently an assistant professor of Computer Science and Engineering at National Sun Yat-sen University in Taiwan, ROC. His research interests include compilers, GPUs, and system modeling.

Wei-Ming Ma https://orcid.org/0000-0002-9334-1415

Wei-Ming Ma was born in Taipei, Taiwan. He received his master's degree (1988) in Hydrographic Sciences from the Naval Postgraduate School, CA, USA. He received his Ph.D. degree (1997) in Physical Oceanography from the Florida Institute of Technology, FL, USA. Dr. Ma was an assistant professor from 2002 to 2014, an associate professor from 2015 to July 2022, has been a professor since August 2022, in the Department of Information Management at Cheng-Shiu University, Taiwan. He has been teaching Introduction to Computer Science at the R.O.C. Air Force Academy since 2016. His research covers SBC architecture, enterprise architecture, information security, digital forensics, and AR/VR multimedia design.

William S. Chao

William S. Chao was born in 1954 in Taiwan and received his Ph.D. degree in information science from the University of Alabama at Birmingham, USA, in 1988. William worked as a computer scientist at GE Research and Development Center, from 1988 till 1991 and has been teaching at National Sun Yat-Sen University, Taiwan from 1992 till 2019. His research covers: systems architecture, hardware architecture, software architecture, and enterprise architecture. Dr. Chao is a member of the Association of Enterprise Architects Taiwan Chapter and also a member of the Chinese Association of Enterprise Architects.