EVALUATING CONSISTENCY BETWEEN UML ACTIVITY AND SEQUENCE MODELS

Yoshiyuki Shinkawa
Department of Media Informatics, Ryukoku University, 1-5 Seta Oe-cho Yokotani, Otsu, Shiga, Japan

Keywords: UML, software engineering, model consistency, formal methods, process algebra.

Abstract: UML activity diagrams and sequence diagrams describe the behavior of a target domain or a system from different viewpoints. When we use these diagrams for modeling the same matter in an application, these diagrams, or the models written by them, must be consistent from each other. However, the evaluation for the consistency is difficult, since these diagrams have considerably different syntax and semantics. This paper presents a process algebraic approach to evaluating the consistency between these models. CCS (Communicating Sequential Processes) is used as process algebra.

1 INTRODUCTION

UML activity diagrams and sequence diagrams describe a target problem domain or a system from two different contrastive viewpoints. The former mainly focus on the usage and external behavior of the system, whereas the latter deal with the interaction between components within the system and depict the internal behavior of it (Ambler, 2004).

In dynamic or behavioral modeling for software development, one of the most effective uses of these diagrams is to represent a requirement model by activity diagrams, for which the implementation model is created using sequence diagrams. In this approach, both the requirement and implementation models must be consistent, or in other words, there must be no conflicts between these models (Elaasar and Briand, 2004).

However it seems difficult to examine and evaluate the consistency, since both the models are represented in the different notations with different syntax and semantics. In addition, the granularity is often different between these models, which makes the evaluation more complicated.

In order to evaluate the consistency rigorously, we first need to reveal the interrelationship between these two models. This interrelationship can be shown in the form of the correspondence between each model element of these models.

However, even though the above element level interrelationship is revealed, it only suggest structural similarity of the models. For the evaluation of inter-model consistency, we have to identify
1. what the models intend to express,
2. what conditions must be met for the consistency, and
3. what inference rules can be applied to conclude the consistency.

Unfortunately, UML is too vague and complicated to accomplish the above. We need more formalized and simplified way to express the behavior of UML activity and sequence models. Various efforts have been made to formalize UML models, using such formal techniques as logic (France et al., 1997), Petri-nets (Hu and Shatz, 2004), algebraic specification (Favre and Clerici, 1999), and other formal techniques (McUmber and Cheng, 2001).

The above formal approaches can express the behavior and semantics of each individual model rigorously, however few efforts were made to evaluate the inter-model consistency including activity-sequence model consistency (Shinkawa, 2006).

This paper presents a formal approach to evaluating the consistency between a UML activity model and a sequence model. The paper is organized as follows. In section 2, we discuss the relationships between a UML activity model and a sequence model between which the granularity might be different.

1In this paper, we refer to a model composed of UML activity diagram as an activity model, or as a sequence model if it is composed of sequence diagrams.
Section 3 presents a process algebraic representation of these UML models. CCS (Communicating Sequential Processes) (Milner, 1989) is used as process algebra. Section 4 presents how the consistency between an activity model and a sequence model is evaluated using CCS.

2 INTERRELATIONSHIPS BETWEEN ACTIVITY AND SEQUENCE MODELS

UML 2.0 provides us with thirteen different kinds of diagrams, which can be classified into two groups of structural diagrams and behavioral diagrams. The former includes class, object, component, deployment, and package diagrams, whereas the latter includes use-case, sequence, communication, timing, interaction overview, state-machine, and activity diagrams.

Even though each diagram can expresses various target domains from various viewpoints, and there are no definite restrictions on which diagrams to be used for a specific purpose, the activity diagram and sequence diagram perform important roles in behavioral modeling for business applications and enterprise information systems. Both diagrams depict the behavioral or dynamic aspects of a system, however the viewpoints are different, from which the models are created.

An activity model, which is composed of activity diagrams, is created from an external viewpoint. For example, in business applications, they often used to represent business processes or workflows which can be observed externally. On the other hand, a sequence model is usually created from an internal viewpoint, and represents interactions between objects or classes that compose a system. These interactions occur within the system, which cannot be observed externally.

We often create these two models for the same matter in an application, one of which represents the external behavior and the other does the internal behavior. In such case, no conflicts are allowed between them, or they must be consistent from each other. However, the evaluation of the consistency between these models becomes complicated since the model elements and their relationships are considerably different between these models. In addition, the granularity of these models might be different, which makes the evaluation more complicated.

For this evaluation, we first have to reveal the structural interrelationship between these two models which might have the different granularity. Successively we need to define the meaning of the consistency between the models.

2.1 Structural Interrelationships

An activity model represents the behavior of a system as a flow of actions. An action is an atomic unit of the behavior, which is connected with other actions using directed arrows called flows. There are two kinds of flows defined, namely control flows and object flows, which represent simple control sequences and message passing respectively.

In addition to these two basic model elements, the following complementary elements are provided:

- Initial and final nodes: represent the starting and ending points of an activity model respectively.
- Fork and join nodes: A fork node splits a single flow into multiple parallel flows, while a join node synchronizes multiple parallel flows into a single flow.
- Decision and merge nodes: A decision node implements a conditional branch, which splits single flow into multiple flows with conditions. Each flow becomes active only when the associated condition is true. A merge node consolidates the above split flows.
- Accept event action and send signal actions: An event and a signal represent interactions with external participants, e.g. people, systems, or processes. An accept event action is an action that waits for such events, while a send signal action is an action that generates and sends a signal according to the events. Events and signals may occur asynchronously.
- Exception handlers: An exception handler deals with exceptional events that occur within the activity.
- Data store nodes: A data store node represents a place for persistent data, e.g. a database system.
- Expansion regions: An expansion region includes action flows inside, and processes an input collection like an array. The region is processed in one of three modes, iteration, parallel, or stream.
- Interruptible activity regions: An action within these regions can be interrupted by an event followed by an action that handles it.

An activity model that is composed of the above elements can be divided into partitions, in order to emphasize the actors or participants that are responsible to the activity, or to emphasize the functionality that each partition performs.
Unlike an activity model, a sequence model represents the behavior of a system as interactions between objects, classes, or actors. The simplest form of a sequence model is made up of a set of lifelines along with the messages that flow between them. A point on a lifeline, from which a message is leaving, is called a sending event occurrence, while a point at which a message is arriving is called a receiving event occurrence. There are several types of messages defined, namely synchronous messages and their return values, asynchronous messages, creation messages, lost messages, and found messages.

In addition to the above basic elements, the following supplementary model elements are defined.

- Execution occurrences: An execution occurrence is a thin rectangle on a lifeline, which indicates that the lifeline is active. If the lifeline represents an object or class, it means the execution of methods.
- Interaction occurrences: An interaction occurrence represents a reference to another sequence model.
- Gates: A gate is used to connect a message to a port outside the current sequence model.
- State invariants: A state invariant expresses a constraint at a specific point on a lifeline.
- Combined fragments: A combined fragment is used to express complicated control flows, e.g. iterations, concurrency, conditional breaks, and so on. The following types of combined fragments are defined.
  - alt: indicates the if-then-else structure
  - opt: indicates the simple if structure
  - break: indicates the exit from the fragment
  - loop: indicates the iteration structure
  - par: indicates the concurrent or parallel execution
  - seq: indicates the sequence of the messages is imposed only to the same lifeline in this fragment
  - strict: indicates the messages must be processed strictly in the given order in this fragment
  - critical: indicates the fragment is a critical section, that is, it must not be interrupted
  - assert, ignore, consider, neg: these are used as comments to the fragment.

As shown above, an activity model and a sequence model consist of considerably different model elements. One of the most fundamental model elements in these models are an action and a message respectively. Each action in an activity model is performed by either a person or a system. In case of a system, a method in an object or a class is invoked manually or automatically. On the other hand, a message in a sequence model reaches to either an object, a class, or an actor. In case of an object or a class, a method in it is invoked at the receiving event occurrence. Therefore, actions and messages or receiving event occurrences are tied together through methods, as far as they are dealt with objects or classes.

In order to define the structural interrelationships between activity models and sequence models, we first regard actions in the former are interrelated with receiving event occurrences in the latter. This definition implies that the flows in the former are interrelated with the messages in the latter.

Based on this basic interrelationship between these models, other model elements can be interrelated as shown in Table 1.

<table>
<thead>
<tr>
<th>Activity Model</th>
<th>Sequence Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial node</td>
<td>The topmost sending event occurrence</td>
</tr>
<tr>
<td>Final node</td>
<td>The bottommost receiving event occurrence</td>
</tr>
<tr>
<td>Decision/Merge node</td>
<td>alt or opt fragment</td>
</tr>
<tr>
<td>Fork/Join node</td>
<td>par fragment</td>
</tr>
<tr>
<td>Accept event action</td>
<td>gate and a dedicated lifeline</td>
</tr>
<tr>
<td>Send signal action</td>
<td>gate and a message to it</td>
</tr>
<tr>
<td>Exception handler</td>
<td>found message and a dedicated lifeline</td>
</tr>
<tr>
<td>Data store node</td>
<td>A dedicated lifeline for data store</td>
</tr>
<tr>
<td>Expansion regions</td>
<td>Combined fragment</td>
</tr>
<tr>
<td>parallel iterate stream</td>
<td>par loop arguments of a message</td>
</tr>
</tbody>
</table>

There are several elements that are not interrelated, e.g. interruptible activity region in a activity model or state invariant in a sequence model. These are referred to in section 3.

### 2.2 Difference in Granularity

An activity model and a sequence model are often created from different viewpoints, involving different stakeholders. Therefore, classes and the methods within them are differently defined between those
models, with different granularity and functional boundaries.

The word “granularity” is often used vaguely in several contexts. However, in our consistency evaluation, methods in classes or objects are the common units between an activity model and a sequence model, in order to match an action with a receiving event occurrence. Therefore we only focus on the granularity of the methods in this paper. When considering the granularity, there are two different standards to determine it. One is the abstraction level, and the other is the scope of input to be processed.

The granularity based on the abstraction level can place the associated methods in order. The granularity of a method with higher abstraction level is larger or coarser than that with lower abstraction level. For example, in modeling the betting round of poker games, a method “bet(int x)” is more abstract than that of “raise(int x)”, “call()”, and “fold()”. The granularity based on the abstraction level is formally defined as follows.

1. Let $f$ and $g$ be methods that are defined as
   
   $f : P_1 \times \cdots \times P_m (= P) \rightarrow R$
   
   $g : Q_1 \times \cdots \times Q_n (= Q) \rightarrow R$

2. If $\{P_i\} \subseteq \{Q_j\}$ and $\forall x \in P \exists y \in Q \; [f(x) = g(y)]$ hold, the method $g$ has larger granularity than $f$, based on the abstraction level.

The granularity of a method $f$ is denoted by $\gamma(f)$, and $\gamma(f) \preceq \gamma(g)$ represents that the granularity of $g$ is larger than or equals to $f$. We can define the intersection and difference of methods in such case. Assuming $\gamma(f) \preceq \gamma(g)$,

   $f \cap g = f$
   
   $g - f : Q_1 \times \cdots \times Q_n - \{y\} \rightarrow R$

On the other hand, the granularity based on the input scope is defined as follows.

1. Let two methods $f$ and $g$ have the same signature.

2. If $\text{Dom}(f) \subseteq \text{Dom}(g)$ and $\forall x \in \text{Dom}(f) \; [f(x) = g(x)]$ hold, the method $g$ has the larger granularity than $f$.

3. If $\text{Dom}(f) \cap \text{Dom}(g) \neq \emptyset$, $\text{Dom}(f) \nsubseteq \text{Dom}(g)$, $\text{Dom}(f) \nsubsetneq \text{Dom}(g)$, and $\forall x \in \text{Dom}(f) \cap \text{Dom}(g) \; [f(x) = g(x)]$ hold, $f$ and $g$ intersect, and have the different functional boundaries.

The intersection and difference of methods can also be defined in such case as follows.

\[
\begin{align*}
\text{Dom}(f) \cap \text{Dom}(g) & \rightarrow \text{Im}(g) \\
\text{Dom}(g) - \text{Dom}(f) & \rightarrow \text{Im}(g)
\end{align*}
\]

where $\text{Im}(g)$ is the co-domain of $g$.

We can define the common methods between an activity model and a sequence model using the above intersection of methods, when the granularity or function boundaries are different between them. The detail is discussed in section 4.

3 \hspace{1cm} EXPRESSING THE BEHAVIOR USING PROCESS ALGEBRA

As discussed in section 1, an activity model represents the external behavior of a system, while a sequence model represents the internal behavior of it. In other words, a sequence model realizes the corresponding activity model. From this viewpoint, we first define the meaning of consistency as “all the behavior in an activity model must occur in the corresponding sequence model, but not vice versa”. Or in other words, any action sequence in the activity model must occur as a series of event occurrences in the sequence model. However, this definition is still vague, and we need to define the behavioral consistency more rigorously. For this purpose, we introduce process algebra, by which we express the behavior more rigorously and evaluate the consistency accurately.

3.1 \hspace{1cm} Process Algebra and Behavioral Equivalency

Process algebra deals with system behavior as algebraic expressions using rigorously defined calculus rules. There have been several process algebras proposed, which include CCS (Milner, 1989), CSP (Hore, 1985), and ACP (Bergström and Klop, 1985). Among them, we use CCS as a consistency evaluation tool, since it treats behavioral equivalency in depth, and there is a standardized specification language LOTOS (Ardis, 1994) based on CCS. CCS represents each behavior of a system as an algebraic expression called a process behavior expression or process expression in short. These process expressions are defined recursively as shown in the upper part of Table 2. The most primitive unit of a CCS process expression is an action, which transforms a process to another one. CCS regards a system as a transition system between processes caused by the above actions. Axioms or inference rules for these transitions are defined as CCS semantics shown in the lower part of Table 2.
Behavioral equivalency between two processes is defined based on the transitions they can make. Many kinds of equivalence relations can be defined in CCS, however the most useful ones are strong equivalence, weak equivalence, and observation congruence, which are shown in Table 3. Among these equivalency relations, observation congruence seems most suitable for our consistency evaluation since it is defined based on the transitions they can make.

### Table 2: CCS Syntax and Semantics.

<table>
<thead>
<tr>
<th>CCS Syntax</th>
<th>CCS Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E = \mathcal{A} \cup \mathcal{I} (\mathcal{A} : \text{set of names} \ \ \mathcal{I} : \text{set of conames})$</td>
<td>$\sum_{i \in I} E_i$ a Summation (if $I = {0,1}$ then $E_1 + E_2$)</td>
</tr>
<tr>
<td>$\text{Act} = \sum_{i \in I} { i } (i : \text{invisible action})$</td>
<td>$E_1[E_2]$ a Composition</td>
</tr>
<tr>
<td>$X : \text{a set of process variables}$</td>
<td>$E^\prime \ L$ a Restriction</td>
</tr>
<tr>
<td>$\mathcal{X} : \text{a set of process expressions}$</td>
<td>$E[f] \ \text{a Relabelling} \ (f \ \text{is a relabelling function})$</td>
</tr>
<tr>
<td>$\exists \ x \ \text{includes } X \ \text{and contains the following expressions, where } E \ \text{and } E_i \ \text{are already in } \mathcal{X}$</td>
<td>$\mu({x \in E })$ a Recursion</td>
</tr>
</tbody>
</table>

| Table 3: CCS Syntax and Semantics. |

<table>
<thead>
<tr>
<th>CCS Syntax</th>
<th>CCS Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{E} = \mathcal{A} \cup \mathcal{I}$</td>
<td>$\sum_{i \in I} E_i$ a Summation (if $I = {0,1}$ then $E_1 + E_2$)</td>
</tr>
<tr>
<td>$\text{Act} = \sum_{i \in I} { i } (i : \text{invisible action})$</td>
<td>$E_1[E_2]$ a Composition</td>
</tr>
<tr>
<td>$X : \text{a set of process variables}$</td>
<td>$E^\prime \ L$ a Restriction</td>
</tr>
<tr>
<td>$\mathcal{X} : \text{a set of process expressions}$</td>
<td>$E[f] \ \text{a Relabelling} \ (f \ \text{is a relabelling function})$</td>
</tr>
<tr>
<td>$\exists \ x \ \text{includes } X \ \text{and contains the following expressions, where } E \ \text{and } E_i \ \text{are already in } \mathcal{X}$</td>
<td>$\mu({x \in E })$ a Recursion</td>
</tr>
</tbody>
</table>

### Table 3: Equivalency between CCS Expressions.

<table>
<thead>
<tr>
<th>Strong Bismilation and Strong Equivalence</th>
<th>Bismilation and Weakly Equivalence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Let $P$, $Q$ be processes, a binary relation $S \subseteq P \times P$ for all $\alpha \in \text{Act}$</td>
<td>Let $P$, $Q$ be processes, a binary relation $S \subseteq P \times P$ for all $\alpha \in \text{Act}$</td>
</tr>
<tr>
<td>$\forall \alpha (P \overset{\alpha}{\rightarrow} P) \ S Q (Q \overset{\alpha}{\rightarrow} Q'$ and $(P', Q') \in S)$</td>
<td>$\forall \alpha (P \overset{\alpha}{\rightarrow} P) \ S Q (Q \overset{\alpha}{\rightarrow} Q'$ and $(P', Q') \in S)$</td>
</tr>
<tr>
<td>$\forall \alpha (Q \overset{\alpha}{\rightarrow} Q') \ S P (P \overset{\alpha}{\rightarrow} P'$ and $(P', Q') \in S)$</td>
<td>$\exists = \cup S : S \text{ is bismilation}$</td>
</tr>
<tr>
<td>$\sim = \cup S : S \text{ is bismilation}$</td>
<td>$P$ and $Q$ is weakly equivalent if $P \sim Q$</td>
</tr>
</tbody>
</table>

$P$ and $Q$ are observationally congruent, if

$\forall \alpha (P \overset{\alpha}{\rightarrow} P) \ S Q (Q \overset{\alpha}{\rightarrow} Q'$ and $(P', Q') \in S)$

$\exists = \cup S : S \text{ is bismilation}$

we write $P \approx Q$ when $P$ and $Q$ satisfies the above.

### 3.2 Transforming an Activity Model into CCS Expressions.

In a UML activity model, an action is a basic unit of the behavior and can be regarded as a member of $\text{Act}$ (a set of actions) in CCS. Since an action in an activity model is associated with a method with arguments and a return value, it can be denoted by $a(\overline{x})$. On another model elements in an activity model are represented in the form of CCS as follows.

**Initial and Final Nodes.** An action $a$ that is directly connected to the initial node means the first action to be taken in the model, while an action $b$ that is directly connected to the final node means the last action. Therefore the whole model can be denoted by $a.E.b$, where $E$ is an interim process between $a$ and $b$. The process prefix form $E.b$ is not a standard CCS notation, however a CCS based specification language LOTOS introduces this form.

**Fork and Join Nodes.** When an action $a$ is split into $n$ processes $E_1, \ldots, E_n$, which subsequently are merged into an action $b$, the whole process can be expressed as $a.(E_1) \cdots (E_n).b$.

**Decision and Merge Nodes.** In order to represent a decision node, we need to express guard conditions associated with it. However CCS does not provides us with any way to describe the conditions. Therefore we introduce a new property to actions, which associates a condition with an action. A condition $c$ that is associated with an action $a$ is denoted by $a[c]$ in this paper, similarly to LOTOS. If one of the actions $b_1, \ldots, b_m$ is performed after an action $a$ according to the conditions $c_1, \ldots, c_n$, the process is expressed as $a.(b_1[c_1] + \cdots + b_m[c_m])$.

**Accept Event Actions and Send Signal Actions.** An accept event action is an action that receive an
asynchronous message to be processed. This action can be represented as a CCS action $r$, and the whole process is expressed using the CCS composition as $E_1 | r . E_2$, where $E_1$ is a process for the normal case and $E_2$ is a process to handle the asynchronous message. On the other hand, a send signal action creates an instance of a signal, which is transmitted to another action called receive signal action. These send/receive actions are expressed as CCS actions associated with a name and its co-name. The action with the co-name does not occur in the CCS expression if the receive signal action belongs to another activity model.

**Exception Handler.** An exception handler is an action that deals with exceptional events. An exceptional event may occur synchronously or asynchronously according to the type of the exception. A synchronous exception handler is expressed similarly to an action after a merge node. On the other hand, in case of an asynchronous exception, there are two possible behaviors that the process shows. One is the behavior that the exception handler interrupts the normal process to terminate it. The other is the behavior that the handler is concurrently scheduled with the normal process. Assuming the exception handler is $e$, the former is expressed in CCS as $\sum E_i . e$ where $E_i$ is a possible process before the exception, while the latter is expressed as $E | \sum E_i . e$, where $E$ represents the normal process.

**Data Store Nodes.** A data store node itself is not a part of the behavior, however the access to the data store node can be regarded as an action to be included in the process.

**Expansion Regions.** An expansion region deals with a list or array of input data in either iterative, parallel, or stream mode. An iterative mode represents a while loop, and can be expressed as a recursion $\mu x . [s \equiv E . (x + s)]$, where $E$ is the process within the region and $s$ is the first action after the region. A parallel mode represents a concurrent processing, and expressed as $E \cdot \cdots \cdot E$. A stream mode represents that all the input elements are processed at the same time, and can be simply expressed as $E$ in CCS.

**Interruptible Activity Region.** An interruptible activity region represents a region within which any actions can be interrupted by other events. The behavior of this region resembles an asynchronous exception handler, and can be expressed as $\sum E_i . e$, where $e$ is an event handler action.

### 3.3 Transforming a Sequence Model into CCS Expressions

In a UML sequence model, a message or its receiving event occurrence corresponds to an action in an activity model and an action in CCS expressions. When synchronous messages $m_1, \ldots, m_p$ flow in the model in this order, the process is expressed as a series of action prefixes $\alpha.m_1 \ldots m_p$, where $\alpha$ is the initial action of the sequence model, which represents the execution occurrence that includes $m_1$.

On the other hand, an asynchronous message splits the process into two concurrent processes, and therefore the process is expressed as $E_1.(E_2[a].E_3)$, where $E_1$ is the process before the asynchronous message $a$, $E_2$ is the normal process after $a$, and $E_3$ is a yielded process by $a$.

More complicated control structures provided by combined fragments are expressed in the form of CCS as follows.

**Alt Fragments and Opt Fragments.** An alt fragment represents the if-then-else structure, and can be expressed in CCS using guard conditions as $a_1[c_1].E_1 + a_2[c_2].E_2$, where $a_i$ is the first action in the upper or the lower section of the fragment, and $c_i$ is the guard condition that activate the upper or lower section. $a_i.E_i$ represents the process within the section. An opt fragment represents a simple if structure, and can be expressed as $a[c].E$.

**Loop Fragments.** A loop fragment is expressed as $\mu x . [E . (x|c|s|\neg c)]$, where $x$ is a process variable that represents the fragment, $E$ is the process expression within the fragment, $c$ is the guard condition for the loop, and $s$ is the first action taken after the fragment.

**Break Fragment.** A break fragment represents a process to terminate the outer fragment to which this fragment directly belongs. This fragment forms an exit point of the outer fragment. The break fragment is associated with a condition $c$ for terminating the outer process, and the outer fragment is expressed as $E_1.(E_2[c].0 + E_3[\neg c])$, where $E_1$ is the process before the break fragment, $E_2$ is the process within the break fragment, and $E_3$ is the process after the break fragment.

**Par Fragments.** A par fragment represents a concurrent execution of processes and simply expressed in CCS using the process composition as $E_1 \cdots E_n$, where $E_i$ is the process within each section in the fragment.
Critical Fragments. A critical fragment represents a process that halts all other processes outside the fragment. In order to implement this fragment, we need a lock mechanism in CCS. For this purpose, we introduce a special variable \( l \) that represents a lock, and special actions lock\((l)\) and unlock\((l)\) which set and reset the lock respectively. The critical fragment is expressed as lock\((l)\).unlock\((l)\), while all the other processes that are concurrently performed with this fragment are associated the condition “the lock \( l \) is reset”.

Seq Fragment. A seq fragment represents that the sequence of messages must be arranged along the lifelines in CCS expressions. Therefore they also are treated as a comment in CCS.

Strict Fragments. A strict fragment represents that the sequence of messages must be arranged along the functional boundaries as discussed in section 2.2. These models must have the same granularity and must be taken into account.

Other Fragment Types. Other fragment types like assert, ignore, etc. are used to make comments to a sequence model. Therefore they also are treated as comments in CCS expressions.

In addition to the above model elements, there are several supplementary elements, namely, gates and state invariants. A gate is used to send a message to, or to receive a message from an object outside the model. A state invariant is a condition that must be satisfied after a specific message, however it can be treated as a comment in CCS.

4 CONSISTENCY EVALUATION

Once an activity model and a sequence model are expressed in the form of CCS process expressions, we can evaluate the equivalency between them using the observation congruence relation as discussed in section 3.1. However, there are several considerations to be taken into account.

Firstly, all the corresponding actions between these models must have the same granularity and functional boundaries as discussed in section 2.2. Secondly, there might be the actions that occur in only one model type, e.g. an action associated with human interactions would occur only in the action model, whereas an action associated with the internal codes of a system would occur only in the sequence model.

In order to adjust these differences between the models, the process expressions may have to be reconstructed using the communized actions based on the communized methods that were introduced in section 2.2. Assuming there are two sets of CCS expressions \( E = \{E_1, \ldots, E_m\} \) and \( F = \{F_1, \ldots, F_n\} \) that are derived from an activity model and a sequence model respectively, this reconstruction can be performed in the following way.

1. Let \( \{a_1, \ldots, a_p\} \) and \( \{b_1, \ldots, b_q\} \) be the actions which are included in \( E \) and \( F \) respectively.
2. For each action \( a_i \) in the activity models, identify a set of activities \( \{b_j\} \) in the sequence model that satisfies
   \[ \forall j, k \ (j \neq k) \ [a_i \cap b_j \neq \emptyset, b_j \cap b_k = \emptyset] \]
3. Define a new action \( a_{ij} = a_i \cap b_j \). If there is no such \( b_j \), replace \( a_i \) by the CCS invisible action \( \tau \).
4. Decompose \( E_i(a_1, \ldots, a_p) \) into a set of expressions \( \{E_{ij}(a_{1j}, a_{2j}, \ldots, a_{nj})\} \) and \( E_i(\hat{a}_1, \hat{a}_2, \ldots, \hat{a}_p) \), where \( \hat{a}_i = a_i - \bigcup_j (a_i \cap b_j) \).
5. Similarly, decompose \( F_i(\hat{b}_1, \ldots, \hat{b}_q) \) into \( \{F_i(\hat{b}_{1j}, \hat{b}_{2j}, \ldots, \hat{b}_{kj})\} \)
6. Each \( E_i(a_1, \ldots, a_p) \) and \( F_i(\hat{b}_1, \ldots, \hat{b}_q) \) are expressed using the common actions \( \{a_{ij}\} \) as 
   \[ E_i = \sum_{j_1 < p} E_i(a_{1j_1}, \ldots, a_{pj}) + E_i(\hat{a}_1, \ldots, \hat{a}_p) \]
   \[ F_i = \sum_{j_1 < q} F_i(\hat{b}_{1j_1}, \ldots, \hat{b}_{qj}) + F_i(\hat{b}_1, \ldots, \hat{b}_q) \]

Following the above steps, the CCS expressions for the activity model and sequence model include the set of common activities \( \{a_{ij}\} \) and the exclusive actions \( \{\hat{a}_i\} \) or \( \{\hat{b}_j\} \), along with the invisible action “\( \tau \)”.

Since \( \{\hat{a}_i\} \) and \( \{\hat{b}_j\} \) are not interrelated between the models, they are to be replaced by “\( \tau \)”.

Even after communizing the actions between the models, there are two more considerations in our approach. One is the guard conditions that are not included in the original CCS definition. The other is the process prefix or the sequential composition \( E_1.E_2 \) that also are not included originally. Since there is no procedure provided to evaluate the equivalency between the process expressions with these elements, we have to develop a procedure.
A process prefix $E_1.E_2$ can be transformed into a regular CCS expression as follows.

1. Identify the final elements $\{A_i\}$ in $E_1$. A final element is either an action $a_j$ that occurs in the form of $a_j.0$ or a recursion $\mu x.[x=E]$.

2. Identify the initial elements $B_k$ in $E_2$. A initial element is either an action with no prefix or a recursion with no prefix.

3. $E_1.E_2$ is transformed into $\sum_{i,j} \varepsilon_i . A_i . B_j . \tau_j$ where $\varepsilon_i$ is the process that ends with $A_i$ and $\tau_j$ is the process that begins with $B_j$.

If $A_i$ is a recursion, it is expressed as $\mu x.[x=E.(x+0)]$ in our approach, and $E.(x+0)$ can be decomposed in the above way. In case of $B_j$, it is expressed as $\mu x.[x=E.(x+E)]$ and can be treated similarly. As for the guard conditions, we can introduce the observation congruence with a condition “c”, which is denoted by $P \equiv_\alpha Q$ as follows.

$$\forall \alpha (P \xrightarrow{\alpha} P') \exists Q (Q \xrightarrow{\alpha} Q' \text{ and } P' \approx Q')$$

In the above definition, $\alpha[c]$ means that the action $\alpha$ can be taken only under the condition $c$. This conditional transition can also be applied to the CCS semantics in Table 2.

Ideally, one CCS expression in the activity model corresponds to that in sequence model. However, a behavior in the activity model may correspond to multiple behaviors in the sequence models, or vice versa. In such case, we need to decompose CCS expressions into appropriate sizes. This decomposition is performed by extracting a continuous part of an expression.

5 CONCLUSIONS

A process algebraic approach to evaluating consistency between a UML activity model and a sequence model is proposed. The behavior of an activity model is characterized by a series of actions, while that of a sequence model is characterized by a series of messages. The behavior of these models must be consistent if they represent the same problem domain or system. The consistency can be evaluated using CCS, a kind of process algebra, if the behavior is expressed in the form of CCS process expressions.

For this evaluation, the granularity and functional boundaries of corresponding actions must be equalized between these models. The equalization is performed by a set theoretic adjustment. Even though activity and sequence models show very different appearances, CCS can express them in the unified form. The behavioral equivalency, which forms the foundation of the consistency between models, is evaluated using CCS observation congruence relation. This relation is extended to deal with conditions that are introduced to represent the complicated control flows in UML models. Similar to the differences in each action, each CCS expression might have the different granularity from each other. It can be resolved by decomposing the expressions.

REFERENCES


