Towards the Formal Analysis of UML Activity Diagrams in a Calculus of Context-aware Ambients

François Siewe
Cyber Technology Institute (CTI)
De Montfort University
Leicester, UK
fsiewe@dmu.ac.uk

***Abstract*—The Unified Modeling Language (UML) is the industrial de-facto standard for designing systems. It has been used widely in many industrial applications. However, the lack of formal semantics for UML makes it unsuitable for formal verification. As such, UML is limited when it comes to the design of safety/security critical systems where faults can cause damages to people, properties, or the environment. This paper proposes an attempt to define a formal semantics for the UML activity diagrams. An algorithm is proposed that translates an activity diagram into a process in a Calculus of Context-aware Ambients (CCA). This process can then be formally analysed using the tool support for CCA. Hence, errors can be detected and fixed early during the system development life-cycle. The pragmatics of the proposed approach is demonstrated using a case study in e-commerce.**

***Index Terms*—UML, activity diagram, CCA, formal analysis, semantics, process calculus, context-awareness, ccaPL**

I. Introduction

The Unified modeling Language (UML) [1] is a graphical language commonly used to specify, visualise, construct, and document the artifact of software-intensive systems. Due to its high expressive power, UML is considered as the industrial de-facto standard for system modeling and was approved as an ISO[1] standard in 2005 [2]. UML provides a standard notation for many types of diagrams which can be roughly divided into 3 main groups, covering all aspects of system design: behavior diagrams, interaction diagrams, and structure diagrams.

Structure diagrams represent the static aspects of the system. They emphasize the things that must be present in the system being modeled, e.g. a class diagram. Interaction diagrams emphasize the flow of control and data among the things in the system being modeled, e.g. a sequence diagram. Behavior diagrams represent the dynamic aspect of the system, i.e. what must happen in the system being modeled. They are used extensively to describe the functionality of software systems. The activity diagram is a UML behavior diagram, which describes the business and operational step-by-step activities of the components in a system.

Although the graphical nature of UML makes it an appealing language for the specification and design of complex software systems, UML (behavior) diagrams lack formal semantics and therefore cannot be formally analysed to ascertain the correctness of the system being modeled. This is particularly important when dealing with safety/security critical systems, where faults in the system can cause harm or damage to people, properties and businesses. Hence, this paper proposes a formal semantics for the UML activity diagram, which maps any activity diagram to a process in the Calculus of Context-aware Ambients (CCA) [3]. CCA is a process calculus that was designed to specify and reason about the behavior of context-aware and mobile concurrent systems. The contributions of this work are threefold:

- An *algorithm* is proposed that translates an activity diagram into an executable process in the Calculus of Context-aware Ambients (CCA). This process can then be formally analysed using the tools support for CCA such as the simulation tool ccaPL and the model-checking tool ccaSPIN. Hence, errors can be detected and fixed early during the system development life-cycle (Sect. IV-B).
- This algorithm is implemented into a Python program that automatically generates a CCA process for any UML activity diagram, at a click of a button (Sect. IV-C).
- The pragmatics of the proposed approach is demonstrated using a case study in e-commerce (Sect. V).

II. Overview of UML Activity Diagrams

A UML Activity diagram models the dynamic aspects of a system, depicted as a flow of control from activity to activity. An activity represents a coordinated flow of actions that the system must perform [2]. A typical UML activity diagram is shown in Fig. 1. Nodes are called activity nodes and the arcs between the nodes are transitions. An *action node* is rendered as round-cornered rectangle. The flow of control of an activity diagram begins in the *initial node* rendered as a solid ball and terminates in a *final node* rendered as a solid ball inside a circle. Swimlanes are used to partition the activity nodes between the actors responsible for executing them. A swimlane boundary is rendered as a vertical solid line, while a transition is rendered as a directed line indicating that the flow of control passes from one activity node to another activity node.

Special activity nodes called fork and join are used to synchronize parallel flows of control and are rendered as a thick horizontal line. A *fork node* has one incoming transition and two or more outgoing transitions, each of which represents an independent flow of control. A *join node* has two or more incoming transitions and one outgoing transition. Alternate

[1]International Organization for Standardization

979-8-3503-2697-0/23/$31.00 ©2023 IEEE
DOI 10.1109/COMPSAC57700.2023.00261

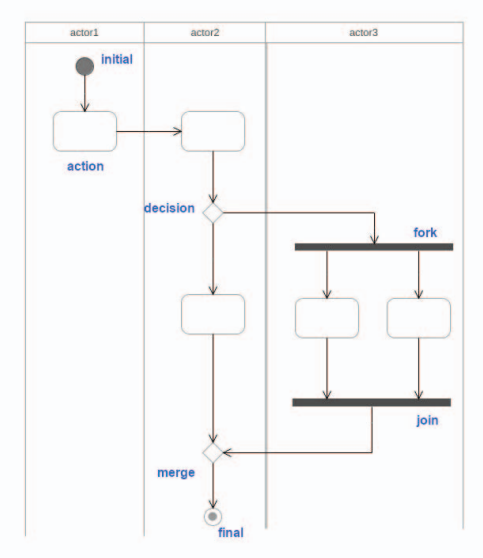


Fig. 1. Elements of an activity diagram

flows based on Boolean expressions can be specified using a *decision node* rendered as a diamond. A decision node has one incoming transition and two or more outgoing transitions; each outgoing transition carries a Boolean expression which must hold for that branch to be taken. These guards across outgoing transitions should not overlap and should cover all possibilities. A *merge node* is also rendered as a diamond and brings together multiple incoming alternate flows to accept a single outgoing flow. Unlike a join node, there is no synchronization of the alternate flows.

There are other activity nodes like object nodes and activity edges that are not included in the scope of this paper, but can be handled in a similar manner.

III. Overview of CCA

The main features of CCA include context-awareness, mobility, and concurrency. The full syntax and semantics of CCA can be found in [3]. Table I depicts the syntax of CCA that is relevant for this paper, based on three syntactic categories: processes P (or Q), capabilities M, and context-expressions κ. The symbols n, x, y and z are names.

Processes: The process $\mathbf{0}$, aka *inactivity process*, does nothing and terminates immediately. The process $P|Q$ denotes the parallel composition of the processes P and Q. The process $(\texttt{new}\ n)\ P$ creates a new name n and the scope of that name is limited to the process P. The replication $!P$ denotes a process which can always create a new copy of P, i.e. $!P$ is equivalent to $P|!P$. The process $n[P]$ denotes an ambient named n whose behavior is described by the process P. A process of the form $\{P\}$ behaves just like P. A context-expression κ specifies a property upon the state of the environment. A *context-guarded prefix* $< \kappa > M.P$ is a process that waits until the environment satisfies the context-expression κ, then performs the capability M and continues like the process P. We let $M.P$ denote the process $< \texttt{true} > M.P$. The *selection* $\texttt{if}\ < \kappa_1 > M_1.P_1\ \ldots\ < \kappa_\ell > M_\ell.P_\ell\ \texttt{fi}$ waits until at least one of the context-expressions $(\kappa_i)_{1\leq i\leq \ell}$ holds; then proceeds non-deterministically like one of the processes $< \kappa_j > M_j.P_j$ for which κ_j holds.

Capabilities: Ambients exchange messages using the output capability $\alpha\ \texttt{send}(z_1, \ldots, z_\ell)$ to send a list of names $z_1, \ldots, z_\ell$ to a location α, and the input capability $\alpha\ \texttt{recv}(y_1, \ldots, y_\ell)$ to receive a list of names from a location α into the variables $y_1, \ldots, y_\ell$. The location α can be '@' to mean any parent, 'n@' to mean a specific parent n, '#' to mean any child ambient, 'n#' to mean a specific child n, '::' to mean any sibling, 'n ::' to mean a specific sibling n, or ϵ (empty string) to mean the executing ambient itself. The capability $\texttt{del}\ n$ deletes an empty n (i.e. $n[\mathbf{0}]$).

Context model: In CCA, a context is modeled as a process with a hole in it. The hole (denoted by $\odot$) in a context C represents the position of the process, which C is a context. For example, suppose a system is modeled by the process $P \mid n[Q \mid m[R \mid S]]$. So, the context of the process R in that system is $P \mid n[Q \mid m[\odot \mid S]]$, and that of the ambient named m is $P \mid n[Q \mid \odot]$. A context-expression (CE, for short) is a formula representing some property over the set of all contexts.

Context-expressions: The CE `true` always holds. A CE $n = m$ holds if the names n and m are identical. The CE `this` holds solely for the hole context, i.e. the position of the process evaluating that context-expression. Propositional operators such as `not`, `and` and `or` expand their usual semantics to context-expressions. A CE $\kappa_1|\kappa_2$ holds for a context if that context is a parallel composition of two contexts such that κ_1 holds for one and κ_2 holds for the other. A CE $n[\kappa]$ holds for a context if that context is an ambient named n such that κ holds inside that ambient. A CE `next` κ holds for a context if that context has a child context for which κ holds. A CE `somewhere` κ holds for a context if that context contains a subcontext for which κ holds. For example, the context-expression $has(n)$ defined in Eq. 1 holds for an ambient if that ambient contains a child ambient named n.

$$has(n) = \texttt{somewhere}(\texttt{this} \mid n[\texttt{true}] \mid \texttt{true}) \qquad (1)$$

IV. Translating an Activity Diagram into a CCA Process

A. A Mathematical Representation of an Activity Diagram

An activity diagram can be represented by the tuple (N, T, A, C, mT, mA, mC), where

- N is a non-empty finite set of activity nodes
- $T \subseteq \{(x, y) \mid x, y \in N \text{ and } x \neq y\}$ is a set of transitions
- A is a finite set of actors (i.e. swimlanes)
- C is a finite set of the conditions (i.e. Boolean expressions) carried by decision nodes
- $mT \subseteq \{(x, y) \mid x \in N \text{ and } y \in \{initial, action, fork, join, decision, merge, final\}\}$ is a type assignment to activity nodes
- $mA \subseteq N \times A$ is a mapping of activity nodes to the actors that perform them

TABLE I
SYNTAX OF CCA

P, Q	$::=$	$\mathbf{0} \mid P \mid Q \mid (\texttt{new}\ n)\ P \mid !P \mid n[P] \mid \{P\} \mid <\kappa> M.P \mid \texttt{if}\ <\kappa_1> M_1.P_1\ \ldots\ <\kappa_\ell> M_\ell.P_\ell\ \texttt{fi}$
M	$::=$	$\alpha\ \texttt{recv}(y_1, \ldots, y_\ell) \mid \alpha\ \texttt{send}(z_1, \ldots, z_\ell) \mid \texttt{del}\ n$
α	$::=$	$@ \mid n@ \mid \# \mid n\# \mid :: \mid n :: \mid \epsilon$
κ	$::=$	$\texttt{true} \mid \texttt{false} \mid \texttt{this} \mid n = m \mid n[\kappa] \mid \texttt{not}\ \kappa \mid \kappa_1 \mid \kappa_2 \mid \kappa_1\ \texttt{and}\ \kappa_2 \mid \kappa_1\ \texttt{or}\ \kappa_2 \mid \texttt{next}\ \kappa \mid \texttt{somewhere}\ \kappa$

- $mC \subseteq \{(x, y, z) \mid (x, y, z) \in C \times N \times N$ and $mT(y) = decision$ and $(y, z) \in T\}$ is a mapping of conditions to the decision node branches.

This representation of an activity diagram is used in the following section to devise an algorithm for translating an activity diagram into a CCA process.

B. An Algorithm to Translate an Activity Diagram into a CCA Process

The proposed algorithm takes as input an activity diagram represented like in Sect. IV-A and generates a CCA process. This algorithm can be described in six main steps:

- **Step 1:** Each actor $a \in A$ is modeled as an ambient named after a. The body of the ambient a will be calculated as a string in the variable $BODY[a]$. Initial the body of a is empty.
- **Step 2:** If the activity diagram contains a decision node or a fork node, define a context-expression $\texttt{has}(n) = \{\texttt{somewhere}(\texttt{this} \mid n[\texttt{true}] \mid \texttt{true})\}$. This context-expression holds if there exists a child ambient named n (see Eq. 1).
- **Step 3:** For each decision node $y \in N$, define a context-expression $y(c) = \{c = x\}$, for x such that $(x, y, z) \in mC$, for some $z \in N$. This context-expression holds for the decision branch guarded by the condition x.
- **Step 4:** Start with the initial node i and update the body of the ambient $mA(i)$ as follows. If the initial node i and its successor $T(i)$ are in the same swimlane, append the string "$\texttt{send}(" + i + ").$" to the body of the ambient $mA(i)$; otherwise append the string "$\texttt{send}(" + i + ").$" $+ mA(T(i)) +$ "$:: \texttt{send}().$" to the body of the ambient $mA(i)$.
  If i is the last node in the swimlane $mA(i)$, then append "$0\backslash n$" to the body of the ambient $mA(i)$.
  Set the current node z to $T(i)$, the successor of i if it exists.
- **Step 5:** Repeat until all the nodes are processed. Let x be the predecessor of the current node z.
  Step 5.1: If the current node z is a *final node*, update the body of the ambient $mA(z)$ as follows. If the nodes z and x are in the same swimlane, append the string "$\texttt{send}(" + z + ").0\backslash n$" to the body of the ambient $mA(z)$; otherwise append the string $mA(x) +$ "$:: \texttt{recv}().\texttt{send}(" + z + ").0\backslash n$" to the body of the ambient $mA(z)$.
  Step 5.2: Else if the current node z is an *action node*, update the body of the ambient $mA(z)$ as follows.
  (a) If x is a decision node, then append to the body of $mA(z)$ the string "$< " + x(c) + " >$", where $(c, x, z) \in mC$.
  (b) If z and x are in the same swimlane, append the string "$\texttt{send}(" + z + ").$" to the body of the ambient $mA(z)$. Otherwise: (1) append the string $mA(x) +$ "$:: \texttt{recv}().\texttt{send}(" + z + ").$" to the body of the ambient $mA(z)$; (2) if $mT(x) \in \{decision\}$, append to the body of $mA(x)$ the string "$< " + x(c) + " >$", where $(c, x, z) \in mC$; and (3) if $mT(x) \in \{decision, fork\}$, append the string $mA(z) +$ "$:: \texttt{send}().$" to the body of the ambient $mA(x)$.
  (c) If z and its successor $T(z)$ are not in the same swimlane, append the string $mA(y) +$ "$:: \texttt{send}().$" to the body of the ambient $mA(z)$. If z belongs to a decision branch $(c, d, t) \in mC$ and $T(z)$ is a merge or join node, then append the string "$< " + d(c) + " >$" to the body of the ambient $mA(T(z))$. If $T(z)$ is a merge or join node, append the string $mA(z) +$ "$:: \texttt{recv}().$" to the body of the ambient $mA(T(z))$.
  (d) If the node $T(z)$ is a merge node and is in the same swimlane as the corresponding decision node, append the string "$\texttt{del}" + T(z) + ".0\backslash n$" to the body of the ambient $mA(T(z))$.
  (e) If the node $T(z)$ is a join node and is in the same swimlane as the corresponding fork node, append the string "$\texttt{del}" + T(z) + ".0$" to the body of the ambient $mA(T(z))$. If this is the last incoming flow of the join node, then append de string "$\backslash n$" to the body of the ambient $mA(T(z))$, otherwise append the string "$|\backslash n$".
  (f) If the node $T(z)$ is a merge (resp. join) node and is not in the same swimlane as the corresponding decision (rep. fork) node y, append the string "$\texttt{del}" + T(z) + ".0$" to the bodies of both the ambients $mA(T(z))$ and $mA(y)$. If this is the last branch (resp. flow) of the decision (resp. fork) node, then append de string "$\backslash n$" to the body of the ambient $mA(T(z))$, otherwise append the string "$|\backslash n$". If this is the last branch (resp. flow) of the decision (resp. fork) node, then append de string "$\backslash n$" to the body of the ambient $mA(y)$, otherwise append the string "$\backslash n$" (resp. "$|\backslash n$").
  (g) If z is the last node in its swimlane and the body of $mA(z)$ does not end with a "$0\backslash n$", then append the string "$0\backslash n$" to the body of the ambient $mA(z)$.
  Step 5.3: Else if the current node z is a *join node*, i.e.

$mT(z) = join$, update the body of the ambient $mA(z)$ as follows. Let y be the corresponding fork node to z. If all the incoming flows of control to z have been processed, then

- Append to the body of $mA(z)$ as many copies of the string "| " + z + "[0]\n" as there are incoming flows.
- If z has a successor, append to the body of $mA(z)$ the string "| < `not has`(" + z + ") > ".
- If y and z are not in the same swimlane, append to the body of $mA(y)$ as many copies of the string "| " + z + "[0]\n" as there are incoming flows to z. If in addition the last node in the swimlane of y is not yet processed, then append the string "| < `not has`(" + z + ") > " to the body of the ambient $mA(y)$.
- Perform the steps (c)-(g) above.
- If the node z does not belong to a decision branch, append the string "}\n" to the body of $mA(y)$.

Step 5.4: Else if the current node z is a *merge node*, i.e. $mT(z) = merge$, update the body of the ambient $mA(z)$ as follows. Let y be the corresponding decision node to z. If x is a decision node, then append to the body of $mA(z)$ the string " < " + $x(c)$ + " > ", where $(c, x, z) \in mC$; and if in addition z and x are in the same swimlane, append the string "`del` " + z + ".0\$n$" to the body of the ambient $mA(z)$, otherwise: (1) append the string $mA(x)$ + " :: `recv`().`del` " + z + ".0\$n$" to the body of the ambient $mA(z)$; (2) append to the body of $mA(x)$ the string " < " + $x(c)$ + " > "; and (3) append the string $mA(z)$ + " :: `send`()." to the body of the ambient $mA(x)$.
If all the incoming branches to z have been processed, then

- Append to the body of $mA(z)$ the string "| " + z + "[0]\n".
- If z has a successor, append to the body of $mA(z)$ the string "| < `not has`(" + z + ") > ".
- If y and z are not in the same swimlane, append to the body of $mA(y)$ the string "| " + z + "[0]\n". If in addition the last node in the swimlane of y is not yet processed, then append the string "| < `not has`(" + z + ") > " to the body of the ambient $mA(y)$.
- Perform the steps (c)-(g) above.
- If $T(z)$ is not a merge node, recursively process the continuation flow starting at $T(z)$.
- If all the nodes in the swimlane of z have been processed, append the string "0\$n$" to the body of $mA(z)$ if it does not end with a "0\$n$".
- Append the string "}\n" to the body of $mA(y)$.

Step 5.5: Else if the current node z is a *fork node*, update the body of the ambient $mA(z)$ as follows.

- If x is a decision node, then append to the body of $mA(z)$ the string " < " + $x(c)$ + " > ", where $(c, x, z) \in mC$.
- If the nodes z and x are in the same swimlane then, if x is a decision node then append the string "`skip`.{\$n$" else append the string "{\$n$" to the body of the ambient $mA(z)$.
- If the nodes z and x are not in the same swimlane, append the string $mA(x)$ + " :: `recv`().{\n" to the body of the ambient $mA(z)$; and (1) if x is a decision node, append the string " < " + $x(c)$ + " > " + $mA(z)$ + " :: `send`()." to the body of the ambient $mA(x)$; (2) if x is a fork node, append the string $mA(z)$ + " :: `send`()." to the body of the ambient $mA(x)$.
- Recursively process each of the parallel flows of control of the fork node.
- If the node z does not belong to a decision branch, append the string "}\n" to the body of the ambient $mA(z)$.

Step 5.6: Else if the current node z is a *decision node*, update the body of the ambient $mA(z)$ as follows.

- If x is a decision node, then append to the body of $mA(z)$ the string " < " + $x(c)$ + " > ", where $(c, x, z) \in mC$.
- If the nodes z and x are in the same swimlane then, if x is a decision node then append the string "`skip`.{ `if`\$n$" else append the string "{ `if`\$n$" to the body of the ambient $mA(z)$.
- If the nodes z and x are not in the same swimlane, append the string $mA(x)$ + " :: `recv`().{ `if`\n" to the body of the ambient $mA(z)$ and the string $mA(z)$ + " :: `send`()." to the body of the ambient $mA(x)$.
- If the nodes z and x are not in the same swimlane, append the string $mA(x)$ + " :: `recv`().{ `if`\n" to the body of the ambient $mA(z)$; and (1) if x is a decision node, append the string " < " + $x(c)$ + " > " + $mA(z)$ + " :: `send`()." to the body of the ambient $mA(x)$; (2) if x is a fork node, append the string $mA(z)$ + " :: `send`()." to the body of the ambient $mA(x)$.
- Recursively process each of the branches of the decision node.
- Append the string "`fi`.0\n" to the body of the ambient $mA(z)$.

Step 5.7: Set z to the successor of the last node processed, if it exists.

- **Step 6:** For each actor $a \in A$, append the string "| !`recv`(t).0\n" to the body of the ambient a. The CCA process generated is the parallel composition of all the ambients $a \in A$.

C. Implementation in Python

The algorithm presented in Sect. IV-B was implemented in Python. The data structure described in Sect. IV-A can easily be implemented in Python using the Python rich type

system, which encompasses data types like sets, lists, tuples, and dictionaries. Due to the space limit, the full Python code cannot be described in detail in this paper. The next section showcases the implementation using a case study of a shopping order processing activity diagram.

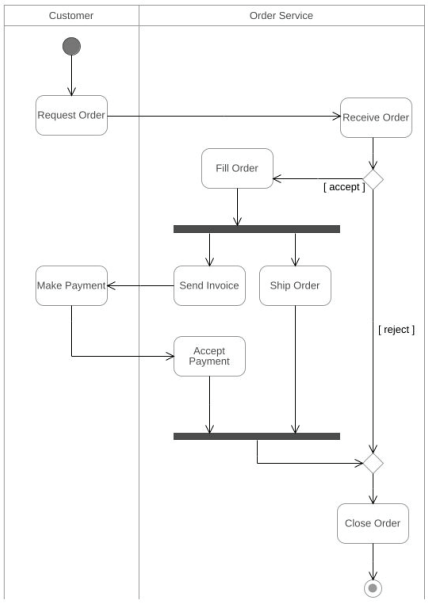


Fig. 2. Shopping order processing activity diagram

V. Case Study: A Shopping order Processing

The activity diagram depicted in Fig. 2 describes the activities involved in the processing of a shopping order in an e-commerce application. This is adapted from the UML 2.4.1 Specification [1]; it contains all the different types of nodes and transitions of a UML activity diagram discussed in Sect. II.

A customer can request an order. After an order is accepted and all required information is filled in, an invoice is sent to the customer. In the meantime, the order shipment is being processed while waiting for the payment to be made by the customer and the payment being accepted by the order service.

This activity diagram is translated into the CCA process in Fig. 3. The declaration section that begins with the keyword `BEGIN_DECLS` and ends with the keyword `END_DECLS` contains the definition of the context-expressions `has(n)` and `decision(c)` as requested in the steps 2 and 3 of the algorithm of Sect. IV-B. Each actor (the customer and the order service) is represented as an ambient in CCA, in accordance with the step 1 of the algorithm. The body of these ambients are constructed according to the steps 4, 5, and 6 of the algorithm.

The generated CCA process in Fig. 3 can now be analysed using the CCA execution tool ccaPL. One can define various scenarios to test the behavior of the activity diagram. Let's consider the scenario where an order request is rejected by the Order Service. To test this scenario, we change `accept` to `reject` in the declaration of the context-expression `decision(c)`. The execution trace output by the ccaPL tool is given in Fig. 4-a. It shows clearly that, the customer submits an order, the order is received and then closed without further processing.

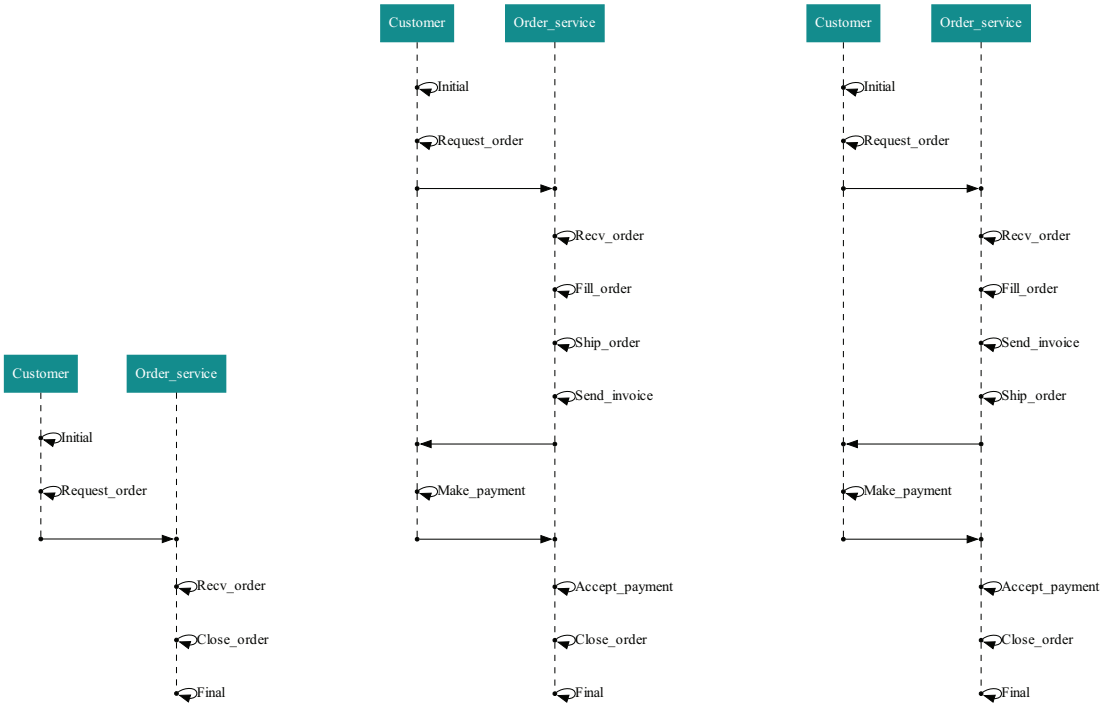
Now assume that the order request is accepted. The execution output for this case is given in Fig. 4-b, showing all the activities that take place from the moment an order is received to shipment and payment. Note that some of these activities are performed in parallel, so their sequential order in the execution trace may vary due to concurrency as illustrated in Fig. 4-c. As one can see, the shipment of an order can take place before the invoice is sent to the customer, and vice versa. This case study demonstrates how the ccaPL tool can be used to analyse the behavior of UML activity diagrams.

```
BEGIN_DECLS
   def has(n) = { somewhere (this | n[true] | true) }
   def decision(c) = { c = accept }
   mode random
END_DECLS
Customer[
   send(Initial).send(Request_order).
   Order_service::send().Order_service::recv().
   send(Make_payment).Order_service::send().0
   | !recv(x).0
]
|
Order_service[
   Customer::recv().send(Recv_order).{ if
      < decision(accept) > send(Fill_order).{
         send(Send_invoice).Customer::send().
         Customer::recv().send(Accept_payment).
         del join.0 | send(Ship_order).del join.0
        | join[0]
        | join[0]
        | < not has(join) > del merge.0
       }
     < decision(reject) > del merge.0
   fi.0
   | merge[0]
   | < not has(merge) > send(Close_order).
   send(Final).0
   }
   | !recv(x).0
]
```

Fig. 3. The generated CCA process for the activity diagram of Fig. 2

VI. Related Work

The authors of [4] investigated the relationship between UML activity diagrams and Petri-nets. They provided a mapping of the basic elements of activity diagrams to Petrinets and discussed the problems arising when trying to extend this approach to some of the advanced features of activity diagrams. This work differs from ours as they are mapping one graphical language to another graphical language. In contrast, our work encodes the behavior of an activity diagram in a process calculus. A formal semantics of UML sequence diagram was proposed in [5], using state transition systems. UML sequence diagrams are one of the UML interaction diagrams, which focus more on the message interchange within a system



(a) Order request is rejected (b) Order request is accepted (c) Yet another order request is accepted

Fig. 4. Examples of execution traces for the activity diagram of Fig. 2

while in contrast, the activity diagrams describe the flows of control that coordinate the activities within the system.

An operational semantics for UML activity diagrams is proposed in [6]. The purpose of their semantics is to give a robust basis for verifying model correctness; to help validate whether a proposed extension of the modeling language is consistent with the standard. Their approach is based on structural operational semantics, while our work proposes a denotational semantics. The two type of formal semantics are complementary to address various types of system properties. The article [7] reviews the various attempts to define a formal semantics for UML and highlighted the difficulties in providing a complete formal semantics for a complex language like UML. We believe that our work contributes to addressing this challenging task.

VII. Conclusion

This paper proposed an algorithm for translating a UML activity diagram into a process in a Calculus of Context-aware Ambients (CCA). This algorithm can be seen as a semantics mapping for UML activity diagrams. The algorithm is implemented in Python to automate the translation process. Therefore, given any UML activity diagram, the corresponding CCA process can be generated at a click of button. It was demonstrated how the generated CCA process can be analysed to understand the behavior of the corresponding activity diagram using the ccaPL tool, the CCA simulator.

In future work, we will extend the proposed approach to more complex modeling languages like the Business Process Modeling Language (BPML). We will also investigate how the CCA model-checking tool ccaSPIN can be used to analyse the properties of activity diagrams, for example the reachability property.

References

[1] *OMG Unified Modeling Language (OMG UML), Superstructure. Version 2.4.1*. Object Management Group, 2011. [Online]. Available: https://www.omg.org/spec/UML/2.4.1

[2] *ISO/IEC 19501:2005 - Information technology - Open Distributed Processing - Unified Modeling Language (UML) Version 1.4.3*, 2005. [Online]. Available: https://www.iso.org/standard/32620.html

[3] F. Siewe, H. Zedan, and A. Cau, "The calculus of context-aware ambients," *Journal of Computer and System Sciences*, vol. 77, no. 4, pp. 597–620, 2011.

[4] H. Störrle and J. H. Hausmann, "Towards a formal semantics of uml 2.0 activities," in *Software Engineering 2005*, P. Liggesmeyer, K. Pohl, and M. Goedicke, Eds. Bonn: Gesellschaft für Informatik e.V., 2005, pp. 117–128.

[5] X. Li, Z. Liu, and H. Jifeng, "A formal semantics of uml sequence diagram," in *2004 Australian Software Engineering Conference. Proceedings.*, 2004, pp. 168–177.

[6] Z. Daw and R. Cleaveland, "An extensible formal semantics for uml activity diagrams," 2016.

[7] M. Broy and M. V. Cengarle, "Uml formal semantics: lessons learned," *Software & Systems Modeling*, vol. 10, pp. 441–446, 2011.