A Formal Framework to Generate XPDL Specifications from UML Activity diagrams
Nicolas Guelfi and Amel Mammar
Software Engineering Competence Center
Faculty of Sciences, Technology and Communication
University of Luxembourg, L-1359 Luxembourg-Kirchberg, Luxembourg
{nicolas.guelfi,amel.mammar}@uni.lu

ABSTRACT

The XML Process Definition Language (XPDL) is a standardized language allowing process definitions interchange between a variety of tools ranging from workflow management systems to modeling and simulation tools. On the other hand, UML activity diagrams offer a convenient notation to depict synthetic and intuitive views of systems that facilitate stakeholder communication. However, there is currently no tool able to animate activity diagrams using workflow management systems. Moreover, despite standardization efforts, diagram interchange across different modeling tools remains an issue. Hence, transforming UML activity diagrams into XPDL specifications would preserve stakeholder communication while enabling animation tool support. This paper presents a formal and optimal transformation of UML activity diagrams into XPDL specifications. This mapping is described through a set of formal translation rules defined from the UML activity metamodel to the XPDL metamodel. We demonstrate that the defined mapping process preserves some structural properties specified on the translated UML activity diagram.

Keywords
UML activity diagrams, XPDL language, Transformations, Correctness proofs.

1. Introduction

The presented work is part of the Efficient project that aims at providing business users with a tool set based on a workflow engine for building trusted business transactions [2]. In this context, UML activity diagrams [11] are used to depict the flow of activities exchanging messages in order to accomplish the underlying transaction [7]. On one hand, the graphical aspects of activity diagrams promote communication between different actors involved in the transaction but, to our knowledge, there is no tool that enables the animation of such specifications using a workflow engine. In addition, there is no identified format for these diagrams that can be used to exchange models built under different environments. On the other hand, XPDL [12] is an XML variant of the standardized process definition language, defined by the Workflow Management Coalition (WfMC), which is compatible with several workflow engines [5] that are well adapted to play scenarios described by an XPDL specification helping, in this way, users have a better comprehension of the processes they are modeling. Furthermore, XPDL is a common interchange format supporting the transfer of workflow process definitions between various workflow systems.

The transformation process, described in this paper, has been designed to be MDA compliant. Introduced by the Object Management Group (OMG) [10], MDA[9] is a new software development paradigm based on model representations. It favors portability, interoperability and reusability by distinguishing several abstraction levels. These levels are connected to each other by abstraction/refinement relationships which are commonly known under the term of “transformations”. Model transformation is the process of transforming a source model $M$ conforming to a given metamodel $MM$ into a target model $M'$ conforming to a metamodel $MM'$ (maybe equal to $MM$). This generic process is the conceptual heart of MDA and other Model Driven Engineering (MDE) approaches [4]. In the MDA vocabulary, source and target models are called PIM (Platform Independent Model) and PSM (Platform Specific Model) respectively, and are defined using the UML notation. Our goal is to define a formal framework in which UML activity diagrams (PIM model) describing business processes are systematically transformed into XPDL specifications (PSM model).

Research work on transforming UML activity diagrams into XPDL or WPDL (an ancestor of XPDL) is somewhat missing. In [6], a mapping of UML activity diagrams, representing declarative collaborative workflow, to XPDL specifications is presented. Although this work covers a large number of UML activity diagram concepts, but no formal rules are defined to support this mapping. Moreover, no guarantee is given about neither the completeness, the termination, nor the correctness of the defined mapping. Bastos and Ruiz [1] consider the reverse mapping, visualizing WPDL with UML activity diagrams, by introducing some annotations on activity diagrams. Similarly to [6], the authors define the mapping in an informal way. Compared to these work, our approach is more rigorous. Based on an informal definition of a transformation approach presented in [3], we define in this article, a precise set of formal translation rules that produce an XPDL specification from a UML activity diagram. We also show how our transformation rules preserve structural properties stated on the source activity diagram. Thus our approach allows the animation of activity diagrams using workflow engines and enables proofs on the correctness of the transformation.

The paper is structured as follows. Section 2 describes the subsets of UML activity diagrams and XPDL metamodels we consider. In Section 3, we present a set of formal transformation and optimization rules that permit mapping an activity diagram to
an optimal (concise) XPDL specification, and discuss some interesting properties of this mapping process. Section 4 shows, through an example, that the defined mapping preserves structural properties that can be proved by reusing already established proofs on the translated activity diagram. Finally, we conclude and outline some future work in Section 5.

2. Metamodels for activity diagrams and XPDL

To be formal, a transformation process must act on the metamodels of the source and target models. This is why metamodels for UML activity diagrams and the XPDL language must be constructed. Metamodels define the structure that must be satisfied by each model conforming to it. The goal of this paper is not to define transformation rules that cover the whole concepts of UML activity diagrams but rather to show the feasibility of the approach especially concerning the proof of the property preservation after transformations.

2.1 Activity diagram metamodel

As already mentioned, an activity diagram is used to model the flow (order) in which activities are executed by exchanging messages (See Figure 2). An activity diagram can be viewed as an oriented graph where vertices and arcs correspond to nodes and transitions respectively. Figure 1 shows the metamodel of such an activity diagram. This figure states that a node may be named; it can be either an activity, a decision/merge node, an initial/final node, a fork/join node or a document node. The initial and final nodes represent the starting and the ending points respectively of the underlying transaction. Activities and documents are represented by ovals and rectangles respectively. A fork node specifies that several parallel threads are launched at the corresponding point. At the decision node, one and only one of the outgoing transitions chosen randomly can be passed. A join (resp. merge) node represents an AND (resp. XOR) synchronization point that means that all (resp. at least one of) the incoming transitions must be passed to continue the flow. Each transition has one source and one target node. Each node may be the source or the target of zero to several transitions. In this paper, we don’t consider guards of transitions.

In Efficient project’s context, a UML activity diagram is denoted by a tuple \( AD = \langle Transitions, \text{NameN}, \text{Kind}, \text{SourceT}, \text{TargetT} \rangle \) where:

- \( \text{Transitions} \) denotes the set of transitions of \( AD \),
- \( \text{NameN} \) denotes the set of nodes of \( AD \),
- \( \text{Kind} \) is a function that associates each transition with its kind: \( \text{Kind} \in \text{Nodes} \rightarrow \{ \text{act, deci, init, fin, fork, join, merge} \} \),
- \( \text{SourceT} \) is a function that associates each transition with its source node: \( \text{SourceT} \in \text{Transitions} \rightarrow \text{Nodes} \),
- \( \text{TargetT} \) is a function that associates each transition with its target node: \( \text{TargetT} \in \text{Transition} \rightarrow \text{Nodes} \).

In the context of our project, we impose the following three constraints that cannot be easily expressed graphically:

\( (Y \in \text{Nodes}) \Rightarrow \forall n \in \text{Nodes} \bullet\left(\begin{array}{l}
(\text{Kind}(n) \in \{\text{fork, deci}\} \land \text{card} (\text{SourceT}^{-1}[\{n\}]) \geq 2) \\
(\text{Kind}(n) = \text{fin} \land \text{card} (\text{SourceT}^{-1}[\{n\}]) = 0) \\
(\text{card} (\text{TargetT}^{-1}[\{n\}]) = 1)
\end{array}\right)\)

\( (Y \in \text{Nodes}) \Rightarrow \forall n \in \text{Nodes} \bullet\left(\begin{array}{l}
(\text{Kind}(n) \in \{\text{merge, join}\} \land \text{card} (\text{TargetT}^{-1}[\{n\}]) \geq 2) \\
(\text{Kind}(n) = \text{init} \land \text{card} (\text{TargetT}^{-1}[\{n\}]) = 0) \\
(\text{card} (\text{SourceT}^{-1}[\{n\}]) = 1)
\end{array}\right)\)

\( (Y \in \text{Nodes}) \Rightarrow \forall n \in \text{Nodes} \bullet\left(\begin{array}{l}
(\text{Kind}(n) \in \{\text{doc}\}) \Rightarrow n \in \text{dom}(\text{NameN})
\end{array}\right)\)

2.2 XPDL metamodel

The main elements of the XPDL language are: Package, Application, WorkflowProcess, Activity, Transition, Participant, DataField and dataType. A full and detailed description of the

\[1 \text{ Symbol } \leftrightarrow \text{ denotes a partial function.}\]

\[2 \text{ If } f: X \leftrightarrow Y, \text{ dom}(f) = \{x \in X \exists y : (y \in Y \land f(x) = y) \} \]
XPDL metamodel can be found in [12]. In this paper, we focus on the control-flow perspective. Therefore, we just consider concepts of Activity, Transition and DataField. The metamodel representing these concepts and their relationships is depicted in Figure 3.

![Figure 3. A partial metamodel for XPDL](image)

The subset of XPDL language we consider is constituted of transitions and activity nodes. In order to distinguish XPDL transitions from those of UML activity diagrams, we denote them by arcs. An arc has exactly one source and one target activity nodes. An activity may be named (NameA) and may be source (resp. target) of several arcs. As transitions of the activity diagrams we consider are not guarded, we make the assumption that the XPDL arcs are not guarded too. When an activity has several incoming (resp. outgoing) arcs, attribute In (resp. Out) specifies the intended behavior. Four different semantics can be distinguished. At the entering of an activity, XORJ (resp. ANDJ) semantics means that the related activity becomes active as soon as at least one of (resp. all) its incoming arcs is passed. Similarly, when an activity is left, the semantics XORS (resp. ANDS) signifies that one and only one outgoing arc is (resp. all outgoing arcs are) passed. Finally, an activity may have a datafield. Formally, an XPDL specification is denoted by a tuple XPDLs =< Arcs, Activities, NameA, IsRoute, DataField, In, Out, SourceA, TargetA > where:

- **A**rcs denotes the set of arcs of the XPDL specification. As arcs will be created during the transformation of activity diagrams into XPDL specifications, we consider a super-type ARCS that represents the set of all possible arcs. Therefore, variable Arcs is typed by: Arcs ⊆ ARCS
- **A**ctivities denotes the set of activities of the XPDL specification. For the same reason as arcs, we consider a super-type ACTIVITIES that represents the set of all possible activities. Hence, variable Activities is typed by: Activities ⊆ ACTIVITIES
- **N**ameA is a function that assigns to each activity its possible name: NameA : Activities → String
- **I**sRoute is a function that specifies whether an activity is a route activity or not: IsRoute ∈ Activities → BOOLEAN
- **D**ataField is a function that returns the possible datafield of an activity: DataField ∈ Activities → String
- **S**ourceA is a function that associates each arc with its source activity: SourceA ∈ Arcs → Activities
- **T**argetA is a function that associates each arc with its target activity: TargetA ∈ Arcs → Activities
- **I**n is a function that gives the semantics of the incoming arcs to an activity: In ∈ Activities → {XORJ, ANDJ}
- **O**ut is a function that gives the semantics of the outgoing arcs from an activity: Out ∈ Activities → {XORS, ANDS}

Let us notice that function In (resp. Out) is partial because this information is significant only for activities with several incoming (resp. outgoing) arcs. Functions In and Out are respectively total on the two subsets Activities1 and Activities2 defined by:

\[
\text{Activities}_1 = \{ A ∈ \text{Activities} \mid \text{card} (\text{TargetA}^{-1}([A])) > 1 \} \\
\text{Activities}_2 = \{ A ∈ \text{Activities} \mid \text{card} (\text{SourceA}^{-1}([A])) > 1 \}
\]

As example of XPDL specification, Figure 4 is the XPDL translation of the activity diagram of Figure 2.

![Figure 4. An optimized XPDL representation of transaction “DealingOrder”](image)

In the above figure, an activity is represented by a white rectangle, an activity with a datafield (called also a process activity) by a grey one, a route activity by a rounded rectangle.

The next section aims at describing a systematic approach to transform a UML activity diagram, respecting the metamodel defined in section 2.1, into an XPDL specification conforming to the metamodel defined in section 2.2.

### 3. Translation rules

In this section, we give a set of formal rules that transforms an activity diagram into an XPDL specification. In order to obtain an optimal XPDL specification while defining simple translation rules, our approach comprises two successive steps:

1. A simple translation process from an activity diagram into an XPDL specification,
2. A set of optimization rules acting on the obtained XPDL specification.

The following two subsections are devoted to the description of these two steps.

#### 3.1 From a UML activity diagram into an XPDL specification

To translate a UML activity diagram into an XPDL specification, two phases are required. Our translation process begins by transforming the nodes into XPDL activities. These activities are linked, in a second phase, with arcs that correspond to the transitions of the related activity diagram. The main idea of transforming an activity diagram into an XPDL specification is to match nodes and transitions with activities and arcs respectively as shown in Figure 5.
The initial and final states for a state of the system. Indeed, these components do not change the activity diagram we are translating are not included in the translation process. Let us remark that the components of XPDL counterpart. These functions become total at the end of the translation stages, not each node (resp. transition) has its elements of TransNodes transformed. Henceforth, we write for (TransNodes is a set of nodes (resp. transitions) that have been already transformed. Henceforth, we write for an activity diagram into XPDL activities. Note that these rules can be applied in any order. Also, we have chosen to model the translation of each kind of node by a distinct rule in order to facilitate the definition of the mapping process. However, these different rules can be merged into a single but a more complicated translation rule.

3.1.1 Translating the initial and final nodes
An initial (resp. final) node denotes a node without incoming (resp. outgoing) transitions. It indicates the starting (resp. ending) point of the activity diagram. The final and the initial nodes are translated similarly. Consequently, only the rule that translates an initial node is given. Formally, the initial node is translated by applying this following rule:

\[ p \xrightarrow{init(x)} q \Leftrightarrow \]
\[ x \in \text{Nodes} \land \text{Kind}(x) = \text{init} \land x \notin \text{dom}(p(\text{TransNodes})) \land \]
\[ y \in \text{ACTIVITIES} \land p(\text{Activities}) \land \]
\[ q(\text{Activities}) = p(\text{Activities}) \cup \{y\} \land \]
\[ q(\text{IsRoute}) = p(\text{IsRoute}) \cup \{y, \text{false}\} \land \]
\[ q(\text{TransNodes}) = p(\text{TransNodes}) \cup \{(x, y)\} \]

The above rule states that it deals with an initial node, denoted by \( x \), by translating it into a route activity denoted by \( y \). The condition \( (x \notin \text{dom}(p(\text{TransNodes}))) \) means that the rule can be applied only once. Indeed, after its first application this condition becomes false as \( x \) will belong to \( \text{dom}(q(\text{TransNodes})) \). This condition is very important to ensure the termination of the transformation process.

3.1.1.2 Translating an activity node
A node of an activity diagram denoting an activity is simply translated into an XPDL activity with the same name. Formally, an activity node is translated as follows:

\[ p \xrightarrow{\text{activity}(x)} q \Leftrightarrow \]
\[ x \in \text{Nodes} \land \text{Kind}(x) = \text{act} \land x \notin \text{dom}(p(\text{TransNodes})) \land \]
\[ y \in \text{ACTIVITIES} \land p(\text{Activities}) \land \]
\[ q(\text{Activities}) = p(\text{Activities}) \cup \{y\} \land \]
\[ q(\text{IsRoute}) = p(\text{IsRoute}) \cup \{y, \text{false}\} \land \]
\[ q(\text{TransNodes}) = p(\text{TransNodes}) \cup \{(x, y)\} \]

3.1.1.3 Translating a fork node
A fork node is translated into an XPDL route activity. Formally, a fork node is translated as follows:

\[ p \xrightarrow{\text{fork}(x)} q \Leftrightarrow \]
\[ x \in \text{Nodes} \land \text{Kind}(x) = \text{fork} \land x \notin \text{dom}(p(\text{TransNodes})) \land \]
\[ y \in \text{ACTIVITIES} \land p(\text{Activities}) \land \]
\[ q(\text{Activities}) = p(\text{Activities}) \cup \{y\} \land \]
\[ q(\text{IsRoute}) = p(\text{IsRoute}) \cup \{y, \text{true}\} \land \]
\[ q(\text{TransNodes}) = p(\text{TransNodes}) \cup \{(x, y)\} \]

3 We consider that it is always possible to create a new activity, that is, the set \( \text{ACTIVITIES} \) is not empty. Such a hypothesis ensures that element \( y \) always exists.
3.1.1.4 Translating a document node

With a document node an XPDL activity (a process activity more precisely) is associated. This activity defines a datafield that represents information about the related document. In our context, we have chosen to give this activity and its datafield the same name, the one of the document. Formally, a document is translated as follows:

\[
p \xrightarrow{\text{doc}(x)} q \iff \\
x \in \text{Nodes} \land \text{Kind}(x) = \text{doc} \land x \notin \text{dom}(p(\text{TransNodes})) \land \\
y \in \text{ACTIVITIES} - p(\text{Activities}) \land \\
q(\text{Activities}) = p(\text{Activities}) \cup \{y\} \land \\
q(\text{NameA}) = p(\text{NameA}) \cup \{(y, \text{NameN}(x))\} \land \\
q(\text{IsRoute}) = p(\text{IsRoute}) \cup \{(y, \text{false})\} \land \\
q(\text{DataField}) = p(\text{DataField}) \cup \{(y, \text{NameN}(x))\} \land \\
q(\text{TransNodes}) = p(\text{TransNodes}) \cup \{(x, y)\}
\]

3.1.2 Transition translation

This paragraph describes the transformation of the transitions connecting the nodes of an activity diagram into arcs linking the activities of the corresponding XPDL specification. Several cases can be distinguished according to the source and target nodes. However in case of transitions whose source or target denotes a special node (fork, join, decision, merge), we have to specify in XPDL the semantics of several arcs incoming to/outgoing from an activity. In other words, we have to assign values to attributes In and Out. For a given XPDL activity, its attribute In (resp. Out) is set only at the generation of the second arc incoming (resp. outgoing) to the considered activity. In fact, the semantics doesn’t change when the other possible incoming (resp. outgoing) arcs are generated in turn. Formally, the translation of a transition is expressed by the following rule:

\[
p \xrightarrow{\text{trans}(x, y, r, x', y')} q \iff \\
tr \in \text{Transitions} \land tr \notin \text{dom}(p(\text{TransTrans})) \land \\
x = \text{Source}(tr) \land y = \text{Target}(tr) \land \\
x' = p(\text{TransNodes})(x) \land y' = p(\text{TransNodes})(y) \land \exists ar \bullet \\
\{ar \in \text{ARCS} - p(\text{Arcc}) \land q(\text{Arcc}) = p(\text{Arcc}) \cup \{ar\} \land \\
q(\text{SourceA}) = p(\text{SourceA}) \cup \{(ar, x')\} \land \\
q(\text{TargetA}) = p(\text{TargetA}) \cup \{(ar, y')\} \land \\
q(\text{TransTrans}) = p(\text{TransTrans}) \cup \{(tr, ar)\} \land \\
\exists ar \bullet p(\text{SourceA})(ar') = x' \land \\
(\text{Kind}(x) = \text{fork} \Rightarrow q(\text{Out}) = p(\text{Out}) \cup \{(x', \text{ANDS})\}) \land \\
\exists ar \bullet p(\text{SourceA})(ar') = x' \land \\
(\text{Kind}(x) = \text{dec} \Rightarrow q(\text{Out}) = p(\text{Out}) \cup \{(x', \text{XORS})\}) \land \\
\exists ar \bullet p(\text{TargetA})(ar') = y' \land \\
(\text{Kind}(y) = \text{join} \Rightarrow q(\text{Out}) = p(\text{Out}) \cup \{(y', \text{ANDJ})\}) \land \\
\exists ar \bullet p(\text{TargetA})(ar') = y' \land \\
(\text{Kind}(y) = \text{merge} \Rightarrow q(\text{Out}) = p(\text{Out}) \cup \{(y', \text{XORJ})\})
\]

In the above rule, the use of the "\exists!" symbol means that we are generating the second arc outgoing from (resp. incoming to) activity \(x'\) (resp. \(y'\)).

3.2 Optimization of an XPDL specification

In this section, we present a set of formal rules that apply on an XPDL specification in order to produce a concise representation. Similarly to translation rules, optimization rules are also modelled as a finite automaton whose initial state corresponds precisely to the final state of the translation process. Naturally, the final state is reached when all the optimization rules become inapplicable. A formal definition of this state is given in [8]. Accordingly to the kinds of UML special nodes, we have defined six optimization rules dedicated to the simplification of the XPDL code. For the sake of concision, only two optimization rules are detailed. Other rules are obtained in a similar way and can be found in [8].

- **Init Opt**: this rule removes the XPDL activity corresponding to the initial node. Indeed, the starting point of an XPDL specification is the activity that has no incoming arc. Therefore, the XPDL activity \(x'\) corresponding to the initial node \(x\) of an activity diagram can be removed when its successor activity \(y'\) has only one incoming arc, that is, the arc outgoing from \(x'\). In fact, after removing this arc, \(y'\) will be without incoming arc. Activity \(y'\) becomes the XPDL translation of \(x\). Of course, we can imagine to not translate the initial node \(x\) when its successor is different from join and merging nodes. However, we didn’t choose this alternative because it would complicate our translation rules. For instance, we must specify that the initial node must be translated at the last. Formally, rule **Init Opt** is defined by:

\[
p \xrightarrow{\text{Init Opt}(x', y', ar)} q \iff \\
x \in p(\text{Activities}) \land y \in p(\text{Activities}) \land ar \in p(\text{Arcc}) \land \\
q(\text{SourceA}) = p(\text{SourceA}) \cup \{(ar, x')\} \land \\
q(\text{TargetA}) = p(\text{TargetA}) \cup \{(ar, y')\} \land \\
q(\text{TransTrans}) = p(\text{TransTrans}) \cup \{(tr, ar)\} \land \\
\exists ar \bullet p(\text{SourceA})(ar') = x' \land \\
(\text{Kind}(x) = \text{fork} \Rightarrow q(\text{Out}) = p(\text{Out}) \cup \{(x', \text{ANDS})\}) \land \\
\exists ar \bullet p(\text{SourceA})(ar') = x' \land \\
(\text{Kind}(x) = \text{dec} \Rightarrow q(\text{Out}) = p(\text{Out}) \cup \{(x', \text{XORS})\}) \land \\
\exists ar \bullet p(\text{TargetA})(ar') = y' \land \\
(\text{Kind}(y) = \text{join} \Rightarrow q(\text{Out}) = p(\text{Out}) \cup \{(y', \text{ANDJ})\}) \land \\
\exists ar \bullet p(\text{TargetA})(ar') = y' \land \\
(\text{Kind}(y) = \text{merge} \Rightarrow q(\text{Out}) = p(\text{Out}) \cup \{(y', \text{XORJ})\})
\]
of elements to which $x'$ corresponds in state $p$, and finally all the transitions translated by $ar$, in state $p$, becomes translated by a null-arc (denoted by $\perp_{x'}$) whose source is $y'$.

- **Fork Opt**: this rule removes the XPDL activity corresponding to a fork node when this latter is not preceded by a decision node. In fact, even if fork and decision nodes are both translated into activities with several outgoing arcs, however their semantics are rather different. Indeed, a fork node has an ANDS semantics while a decision node has an XORS semantics. As a result these nodes cannot be merged (See Figure 6).

![Figure 6. Merging a fork node with its antecedent](image)

Formally, the **Fork Opt** rule is specified as follows:

$$p \xrightarrow{\text{Fork Opt}(x',y',ar)} q \iff$$

$$\left\{ \begin{array}{l}
x' \in \text{Activities} \wedge (x' \in \text{dom}(p(\text{Out}))) \Rightarrow (p(\text{Out}))(x') \neq \text{XORS} \wedge \\
y' \in \text{Activities} \wedge y' \in \text{dom}(p(\text{Out})) \wedge (p(\text{Out}))(y') = \text{ANDS} \wedge \\
ar \in \text{Arcs} \times x' \in (p(\text{SourceA})(ar) \wedge y' = (p(\text{TargetA}))(ar)) \wedge \\
q(\text{Activities}) = p(\text{Activities}) - \{y'\} \wedge \\
q(\text{Arcs}) = p(\text{Arcs}) - \{ar\} \wedge q(\text{TargetA}) = \{ar\} \cup \{p(\text{TargetA})\} \wedge \\
q(\text{SourceA}) = (\{ar\} \cup \{p(\text{SourceA})\}) \wedge (\text{SourceA}^{-1}[\{y'\} \times \{x'\}] \wedge \\
q(\text{TransNodes}) = p(\text{TransNodes}) \cup \{p(\text{TransNodes}^{-1}[\{y'\} \times \{x'\}] \wedge \\
q(\text{TransTrans}) = p(\text{TransTrans}) \cup \{p(\text{TransTrans}^{-1}[\{ar\}] \times \{\perp_{x'}\} \wedge \\
q(\text{Out}) = p(\text{Out}) \cup \{x', \text{ANDS}\} \\
\right\}$$

This last formula says that the XPDL activity $y'$ can be merged with its antecedent $x'$ if and only if the semantics of the possible outgoing transitions from $x'$ is different from XORS. In this case, activity $y'$ and arc $ar$ linking it to $x'$ are both removed, tuples related to arc $ar$ are removed from functions $\text{SourceA}$ and $\text{TargetA}$, all the arcs outgoing from $y'$ become outgoing from $x'$, $x'$ becomes the XPDL translation of all the nodes to which $y'$ corresponds in state $p$, all the transitions translated by arc $ar$, in state $p$, become translated by a null-arc whose source is $x'$, and finally, ANDS semantics is set on the out of activity $x$.

### 3.3 Completeness and termination

Completeness and termination are two important properties to be checked for each transformation process. In our framework, the completeness property ensures that each UML activity verifying the syntax defined in section 2.1 can be mapped to an XPDL specification. The termination property guarantees the finiteness of transformations. Our translation process is complete since we have defined a rule for each element of an activity diagram. Moreover, it terminates because one and only one element of a transition in the source activity diagram corresponds to a path composed of exactly one transition. The second one represents the general case. As users are especially interested in documents and activities, we consider that $A$ and $B$ denote either a document or an activity. Our ultimate goal is to prove that in the XPDL specification obtained after translation and optimization steps, it exists a path from the XPDL activity $A'$ to the XPDL activity $B'$ corresponding to $A$ and $B$ respectively. To establish such a property, two phases are required:

1. Proving that in the XPDL specification obtained by the translation step, it exists a path from $A'$ to $B'$ (**Phase 1**).
2. Proving that after possible optimizations, a path from $A'$ to $B'$ exists too. Note that this path may be different from that exhibited in the first phase in sense that it may be constituted of different activities and/or arcs (**Phase 2**).

Let us detail each of these two phases.

**Phase 1**

Our goal is to prove that it exists a path from $A'$ to $B'$ where $A'$ and $B'$ are the XPDL translation of $A$ and $B$ respectively:
In other words, we have to establish \( \text{path}(A', B') \) which is defined by:

\[
\begin{align*}
A' &= \text{TransNodes}(A) \\
B' &= \text{TransNodes}(B)
\end{align*}
\] (1)

Let \( C_0 \) and \( tr_0 \) be the values of \( C \) and \( tr \) that satisfy formula (7). These values satisfy:

\[
\begin{align*}
(C_0 \in \text{Nodes} - \{A, B\} \land & \ tr_0 \in \text{Transitions} \land \\
\text{Source}T(tr_0) &= {}_C A \land \text{Target}T(tr_0) = B \land \\
\text{path}(A, C_0)
\end{align*}
\] (8)

To prove formula (2) different cases must be studied according to the kind of \( C_0 \). In this paper, we illustrate the case of a fork node. Other cases can be dealt with similarly. We consider now, that node \( C_0 \) denotes a fork node. Let us prove that the XPDL elements corresponding to \( C_0 \) and \( tr_0 \) can be used to establish (2). After applying rule \( \text{fork} \) on \( C_0 \), we deduce that (only the relevant conjuncts are kept, the reference to state \( q \) is removed):

\[
\exists C_0' \bullet \left( C_0' \in \text{Activities} \land \text{TransNodes}(C_0) = C_0' \right)
\] (9)

Let \( C_0' \) be the value of \( C_0' \) that verifies (9). So, we have:

\[
\left( C_0' \in \text{Activities} \land \text{TransNodes}(C_0) = C_0' \right)
\] (10)

Similarly, by applying rule \( \text{trans} \) on \( (B, C_0, tr_0, B', C_0' \land \text{fork} \), we deduce that (only the relevant conjuncts are kept, the reference to state \( q \) is removed):

\[
\exists ar_0 \bullet \left( ar_0 \in \text{Trans}(tr_0) \land \right.
\left. \text{Source}A(ar_0) = C_0' \land \text{Target}A(ar_0) = B' \right)
\] (11)

Let \( ar_0 \) be the value of \( ar \) that verifies (11). So, we have:

\[
\left( ar_0 \in \text{Trans}(tr_0) \land \right.
\left. \text{Source}A(ar_0) = C_0' \land \text{Target}A(ar_0) = B' \right)
\] (12)

Hereafter, we prove that \( C_0' \) and \( ar_0 \) can be reused to prove (2). By substituting \( C_0' \) and \( ar_0 \) in the second part of (2), the proof comes down to establish:

\[
\begin{align*}
& \left( C_0' \in \text{Activities} - \{A', B'\} \land \\
& \quad ar_0 \in \text{Trans}(tr_0) \land \\
& \quad \text{Source}A(ar_0) = C_0' \land \\
& \quad \text{Target}A(ar_0) = B' \land \\
& \quad \text{path}(A', C_0'')
\end{align*}
\] (13)

Let us verify each conjunct of predicate (13):

- \( (C_0' \in \text{Activities} - \{A', B'\}) \): to formally prove this predicate, consider states \( p, q, r \) and \( s \) defined as follows:
  1. In state \( p \), nodes \( A \) and \( B \) are not translated yet.
  2. In state \( q \), just node \( A \) has been translated into \( A' \). So, we have:
    \[
    \left( A' \in \text{ACTIVITIES} - p(\text{Activities}) \land \\
    q(\text{Activities}) = p(\text{Activities}) \cup \{A'\} \right)
    \]
  3. In state \( r \), node \( B \) is also translated into \( B' \). So, we have:
    \[
    \left( B' \in \text{ACTIVITIES} - q(\text{Activities}) \land \\
    r(\text{Activities}) = q(\text{Activities}) \cup \{B'\} \right)
    \]
4. In state s, node C₀ is also translated into C₀'. So, we have:
\[
\begin{align*}
    C₀' & \in ACTIVITIES - r(\text{Activities}) \land \\
    s(\text{Activities}) & = r(\text{Activities}) \cup \left\{ C₀' \right\}
\end{align*}
\]
From the last three points, it is clear that (C₀' ∈ Activities - {A', B'}) is fulfilled. A similar proof can be done whatever the order in which nodes A, B and C₀ are translated.

- The second and third conjuncts are also satisfied according to formula (12).
- Using the inductive assumption, the last conjunct is also satisfied since the length of the path from A to C is equal to k.

**Phase 2**

In this paper, we make the assumption that optimization rules are applied one after another. This means that the simultaneous application of several rules is not considered yet. Therefore, to prove that a path from A' to B' exists after possible optimizations, we have just to establish that is true after a possible application of each optimization rule we have defined.

In this paper, we illustrate such a proof with the Fork_Opt rule, but the principle remains the same for other optimization rules. To make rule Fork_Opt applicable, we consider that on (A' \rightarrow B'), there exist three activities C₁, C₂ and C₃, and two arcs ar₁ and ar₂ such that (See Figure 7):

\[
\begin{align*}
    path(A', C₁) \land SourceA(ar₁) &= C₁ \land TargetA(ar₁) = C₂ \land \\
    SourceA(ar₂) &= C₂ \land TargetA(ar₂) = C₃ \land \\
    path(C₃, B') \land Out(C₂) &= ANDS \land \\
    (C₁ \in dom(Out) \Rightarrow Out(C₁) \neq XORS)
\end{align*}
\]

By applying rule Fork_Opt on (C₁, C₂, ar₁), we obtain:

\[
\begin{align*}
    path(A', C₁) \land SourceA(ar₁) &= C₁ \land \\
    TargetA(ar₁) &= C₂ \land path(C₃, B')
\end{align*}
\]

This formula is simply equivalent to path(A', B').

**5. Conclusion**

In this paper, we have presented a systematic approach to generate by transformation XPDL specifications from UML activity diagrams. The defined approach operates in two steps. The first step maps each UML activity element to a new XPDL element. The second step applies a set of optimization rules in order to obtain an optimal XPDL specification. We have also shown how it is possible to reuse already established property proofs on activity diagrams to prove similar properties on the XPDL specification translating them.

Future work includes extending our approach to support more elaborate concepts as those defined in UML 2.0 specification especially concerning nested transactions. We plan also to study how several translation/optimization rules could be combined to be applied simultaneously. It would be particularly interesting to inspect the effect of such rule compositions on the proof reuse as illustrated in section 4.

**6. References**


