An Automated Approach to Transform Use Cases into Activity Diagrams

Conference Paper *in* Lecture Notes in Computer Science · June 2010

DOI: 10.1007/978-3-642-13595-8_26 · Source: DBLP

3 authors:

Tao Yue
Simula Research Laboratory
171 PUBLICATIONS **3,301** CITATIONS

SEE PROFILE

Lionel C. Briand
Simula Research Laboratory
357 PUBLICATIONS **26,667** CITATIONS

SEE PROFILE

Yvan Labiche
Carleton University
177 PUBLICATIONS **7,608** CITATIONS

SEE PROFILE

An Automated Approach to Transform Use Cases into Activity Diagrams

Tao Yue[1], Lionel C. Briand[2], and Yvan Labiche[1]

[1] Carleton University, Software Quality Engineering Lab,1125 Colonel By Drive Ottawa, ON K1S 5B6, Canada
{tyue, labiche}@sce.carleton.ca

[2] Simula Research Laboratory & University of Oslo, P.O. Box 134, Lysaker, Norway
briand@simula.no

Abstract. Use cases are commonly used to structure and document requirements while UML activity diagrams are often used to visualize and formalize use cases, for example to support automated test case generation. Therefore the automated support for the transition from use cases to activity diagrams would provide significant, practical help. Additionally, traceability could be established through automated transformation, which could then be used for instance to relate requirements to design decisions and test cases. In this paper, we propose an approach to automatically generate activity diagrams from use cases while establishing traceability links. Data flow information can also be generated and added to these activity diagrams. Our approach is implemented in a tool, which we used to perform five case studies. The results show that high quality activity diagrams can be generated. Our analysis also shows that our approach outperforms existing academic approaches and commercial tools.

Keywords: Use Case; Use Case Modeling; UML; Activity Diagram; Transformation; Traceability; Automation; Natural Language Processing.

1 Introduction

Use case modeling, through use case diagrams and use case textual specifications, is commonly applied to structure and document requirements (e.g., [15]). In this context, UML Activity diagrams are often used to: 1) Visualize use case scenarios to better understand and analyze them (e.g., [21]), which becomes paramount when use cases are large and complex; 2) Model business processes, work and data flows, which information is embedded in use case descriptions (e.g., [5]); 3) Complement analysis models by providing an additional, complementary view to class and interaction diagrams (e.g., [9]), and 4) Generate test cases complying with use cases (e.g., [17]). Automated support to transform a use case description into an (initial) activity diagram is therefore important.

Additionally, automated transformation would enable automated traceability from requirements to activity diagrams. Traceability is important during software development since it allows engineers to understand the connections between various artifacts of a software system. Traceability is also mandated by numerous standards (e.g., IEEE Std. 830-1998 [1]) to support, for example, safety verification [18].

We conducted a systematic literature review [27] on transformations of textual requirements into analysis models, including class, sequence and activity diagrams.

We also reviewed more recent publications that were not included in our initial systematic review, as well as existing commercial tools. Existing approaches were compared with our tool according to a set of criteria (Section 6). Results show that some approaches are not fully automated, that the automated ones are not necessarily practical and complete, and that most of them do not provide support for traceability.

The basis of our approach is a use case modeling approach RUCM [25], which relies on a use case template and a set of restriction rules for textual Use Case Specifications (UCSs) to reduce the imprecision and incompleteness inherent to UCSs. We have conducted a controlled experiment to evaluate RUCM and results indicate that RUCM, though it enforces a template and restriction rules, has enough expressive power, is easy to use, and helps improve the understandability of use cases and the quality of derived analysis models [25].

The current work is part of the aToucan approach and tool [26], which aims to transform a Use Case Model (UCMod) produced with RUCM into a UML analysis model that includes class, sequence and activity diagrams. aToucan involves three steps. First, requirements engineers manually define use cases by following RUCM [25]. Second, aToucan reads these textual UCSs to identify Part-Of-Speech (POS) and grammatical relation dependencies of sentences, and then records that information into an instance of the metamodel UCMeta (our intermediate model) (Section 2.2). The third step is to transform the instance of UCMeta into an analysis model as an instance of the UML 2.0 metamodel. During these transformations, aToucan establishes traceability links between the UCMod and the generated UML diagrams.

In this paper, we focus on the RUCM to activity diagrams transformation of aToucan. Specifically, aToucan can automatically generate two types of activity diagrams for each use case: A *detailed activity diagram* shows the main use case flow as well as all alternative flows in one activity diagram; An *overview activity diagram*, on the other hand, only details the main use case flow while the alternative flows are detailed in parts of the sequence diagram aToucan generates for the use case. The activity diagram of the main flow refers to parts of the sequence diagram thanks to the UML 2.0 notions of `CallBehaviorAction` and `Interaction`. Overview activity diagrams therefore help to handle complexity in use case descriptions (complex flows, numerous flows). Our approach can also automatically attach data flow information to generated (overview or detailed) activity diagrams. This is useful to measure complexity or facilitate data flow-based testing for instance [24].

Five case studies have been performed to evaluate activity diagrams generated by aToucan. Results show that complete and correct (against UCSs) activity diagrams can be generated and traceability links can also be correctly established. Our study also indicates that aToucan outperforms three commercial tools.

The rest of the paper is organized as follows. Section 2 discusses RUCM and UCMeta. Section 3 discusses our transformation approach. Tool support is briefly discussed in Section 0. Case studies are discussed in Section 5. Section 6 discusses related work. Section 6 concludes the paper.

2 Background

We briefly review the use case modeling approach RUCM (Section 2.1) and the intermediate model (UCMeta) of our transformations (Section 2.2) [25].

2.1 RUCM

RUCM encompasses a use case template and 26 well-defined restriction rules [25]. Rules are classified into two groups: restrictions on the use of Natural Language (NL), and rules enforcing the use of specific keywords for specifying control structures. The goal of RUCM is to reduce ambiguity and facilitate automated analysis. A controlled experiment evaluated RUCM in terms of its ease of application and the quality of the analysis models derived by trained individuals [25]. Results showed that RUCM is overall easy to apply and that it results in significant improvements over the use of a standard use case template (without restrictions to the use of NL), in terms of the correctness of derived class diagrams and the understandability of UCSs. Below we discuss the features of RUCM that are particularly helpful to generate activity diagrams. An example of UCS documented with RUCM is presented in Table 1.

A use case description has one basic flow and can have one or more alternative flows (first column in Table 1). An alternative flow always depends on a condition occurring in a specific step in a flow of reference, referred to as *reference flow*, which is either the basic flow or an alternative flow itself. We classify alternative flows into three types: A *specific alternative flow* refers to a specific step in the reference flow; A *bounded alternative flow* refers to more than one step in the reference flow– consecutive steps or not; A *global alternative flow* (called *general alternative flow* in [3]) refers to any step in the reference flow.

Distinguishing different types of alternative flows makes interactions between the reference flow and its alternative flows much clearer. For specific and bounded alternative flows, a RFS (Reference Flow Step) section specifies one or more (reference flow) step numbers. Whether and where the flow merges back to the reference flow or terminates the use case must be specified as the last step of the alternative flow. Branching condition, merging and termination are specified by following restriction rules that impose the use of specific keywords (see below). By doing so, we can avoid potential ambiguity in UCSs caused by unclear specification of interactions between the basic flow and its alternative flows, and facilitate automated generation of activity diagrams.

RUCM defines a set of keywords to specify conditional logic sentences (IF-THEN-ELSE-ELSEIF-ENDIF), concurrency sentences (MEANWHILE), condition checking sentences (VALIDATES THAT), and iteration sentences (DO-UNTIL). These keywords greatly facilitate the automated generation of activity diagrams as they clearly indicate when alternative flows start and which kind of alternative flow starts, for which there exist a direct mapping to some UML activity diagram notation. For example, concurrency sentences with keyword MEANWHILE can be accurately transformed into a fork node, a join node, and a number of actions between the fork and join nodes corresponding to the parallel sentences connected by keyword MEANWHILE. Keywords ABORT and RESUME STEP are used to describe an exceptional exit action and where an alternative flow merges back in its reference flow, respectively. An alternative flow ends either with ABORT or RESUME STEP, which means that the last step of the alternative flow should clearly specify whether the flow returns back to the reference flow and where (using keywords RESUME STEP followed by a returning step number) or terminates (using keyword ABORT).

Table 1 Use case Withdraw Fund (originally from [11], and written here by applying RUCM)

Use Case Name	Withdraw Fund		
Brief Description	ATM customer withdraws a specific amount of funds from a valid bank account.		
Precondition	The system is idle. The system is displaying a Welcome message.		
Primary Actor	ATM customer	*Secondary Actors*	None
Dependency	INCLUDE USE CASE Validate PIN.	*Generalization*	None
Basic flow steps	**1)** INCLUDE USE CASE Validate PIN. **2)** ATM customer selects Withdrawal. **3)** ATM customer enters the withdrawal amount. **4)** ATM customer selects the account number. **5)** The system VALIDATES THAT the account number is valid. **6)** The system VALIDATES THAT ATM customer has enough funds in the account. **7)** The system VALIDATES THAT the withdrawal amount does not exceed the daily limit of the account. **8)** The system VALIDATES THAT the ATM has enough funds. **9)** The system dispenses the cash amount. **10)** The system prints a receipt showing transaction number, transaction type, amount withdrawn, and account balance. **11)** The system ejects the ATM card. **12)** The system displays Welcome message. **Postcondition**: ATM customer funds have been withdrawn.		
Specific Alt. Flow (RFS Basic flow 8)	**1)** The system displays an apology message MEANWHILE the system ejects the ATM card. **2)** The system shuts down. **3)** ABORT. **Postcondition**: ATM customer funds have not been withdrawn. The system is shut down.		
Bounded Alt. Flow (RFS Basic flow 5-7)	**1)** The system displays an apology message MEANWHILE the system ejects the ATM card. **2)** ABORT. **Postcondition**: ATM customer funds have not been withdrawn. The system is idle. The system is displaying a Welcome message.		
Global Alt. Flow	IF ATM customer enters Cancel THEN **1)** The system cancels the transaction MEANWHILE the system ejects the ATM card. **2)** ABORT. ENDIF **Postcondition**: ATM customer funds have not been withdrawn. The system is idle. The system is displaying a Welcome message.		

2.2 UCMeta

UCMeta is the intermediate model in aToucan [26], used to bridge the gap between a textual UCMod and a UML analysis model (class, sequence and activity diagrams). As a result, we have two transformations: from the textual UCMod to the intermediate model, and from the intermediate model to the analysis model. Metamodel UCMeta also complies with the restrictions and use case template of RUCM. The current version of UCMeta is composed of 108 metaclasses and is expected to evolve over time. The detailed description of UCMeta is given in [26].

UCMeta is hierarchical and contains five packages: `UML::UseCases`, `UCSTemplate`, `SentencePatterns`, `SentenceSemantics`, and `SentenceStructure`. `UML::UseCases` is a package of UML 2 superstructure [19], which defines the key concepts used for modeling use cases such as actors and use cases. Package `UCSTemplate` models the concepts of the use case template of RUCM: those concepts model the structure that one can observe in Table 1. `SentencePatterns` is a package describing different types of sentence patterns, which uniquely specify the grammatical structure of simple sentences, e.g., `SVDO` (subject-verb-direct object) (Table 1, Basic flow, step 2). `SentenceSemantics` is a package modeling the classification of sentences from the aspect of their semantic functions in a UCMod. Each sentence in a UCS can either be a `ConditionSentence` or an `ActionSentence`. Package `SentenceStructure` takes care of NL concepts in sentences such as subject or Noun Phrase (NP). Package `UCSTemplate` is mostly related to the activity diagram generation and therefore it is

the only package discussed below due to space limitation.

Package `UCSTemplate` not only models the concepts of the use case template but also specifies three kinds of sentences: `SimpleSentence`s, `ComplexSentence`s, and `SpecialSentence`s. In linguistics, a `SimpleSentence` has one independent clause and no dependent clauses [4]: one `Subject` and one `Predicate`. UCMeta has four types of `ComplexSentence`s: `ConditionCheckSentence`, `ConditionalSentence`, `IterativeSentence`, and `ParallelSentence`, which correspond to four keywords that are specified in RUCM (Section 2.1) to model conditions (IF-THEN-ELSE-ELSEIF-THEN-ENDIF), iterations (DO-UNTIL), concurrency (MEANWHILE), and validations (VALIDATES THAT) in UCS sentences. UCMeta also has four types of special sentences to specify how flows in a use case or between use cases relate to one another. They correspond to the keywords RESUME STEP, ABORT, INCLUDE USE CASE, EXTENDED BY USE CASE, and RFS (Reference Flow Step).

3 Approach

Recall that our objective is to automatically transform a textual UCMod expressed using RUCM into UML activity diagrams while establishing traceability links. We present an overview of our approach in Section 3.1 and then detail transformation rules (Section 3.2), transformation algorithm (Section 3.3) and traceability (Section 3.4).

3.1 Overview

In this section, we use Fig. 1 as a running example. It shows a piece of the use case description of Table 1: Fig. 1 (a). The first transformation is to automatically transform a textual UCMod (Fig. 1 (a)) into an instance of UCMeta (Fig. 1 (b)) through a set of transformation rules. For example, basic flow step 8 of Fig. 1 (a) is transformed into an instance of `ConditioncheckSentence` (Fig. 1 (b)). Notice in Fig. 1 (b) that this `ConditioncheckSentence` instance is linked to a `BasicFlow` instance (step 8 is part of the basic flow in Table 1) of the `UseCaseSpecification` of `UseCase Withdraw Fund`. Fig. 1 (b) does not show how the sentence of step 8 is further transformed into instances of UCMeta (e.g., verb, subject).

Second, the UCMeta instance is automatically transformed into a UML analysis model through another set of transformation rules. The (UML 2.0) analysis model contains a class diagram, and a sequence and an activity diagram for each use case. Generating class and sequence diagrams is discussed in [26]. In this paper, we particularly focus on the transformation to activity diagrams.

As mentioned earlier, two types of activity diagrams can be generated for a use case, i.e., from an instance of UCMeta. A *detailed activity diagram* shows the main use case flow as well as all alternative flows in one activity diagram: Fig. 1 (d); whereas an *overview activity diagram* only details the main use case flow while the alternative flows are detailed in parts of the sequence diagram generated for the use case (instances of `Interaction`): Fig. 1 (c). To illustrate the difference, first note that the parts of Fig. 1 (c) and (d) highlighted with rectangles detail the main flow of the use case in the same way: one can recognize step 8 ("`The system VALIDATES THAT` …") followed by a decision node (the validation may be successful or not). In the detailed activity diagram (Fig. 1 (d)) the alternative flow (i.e., when the validation

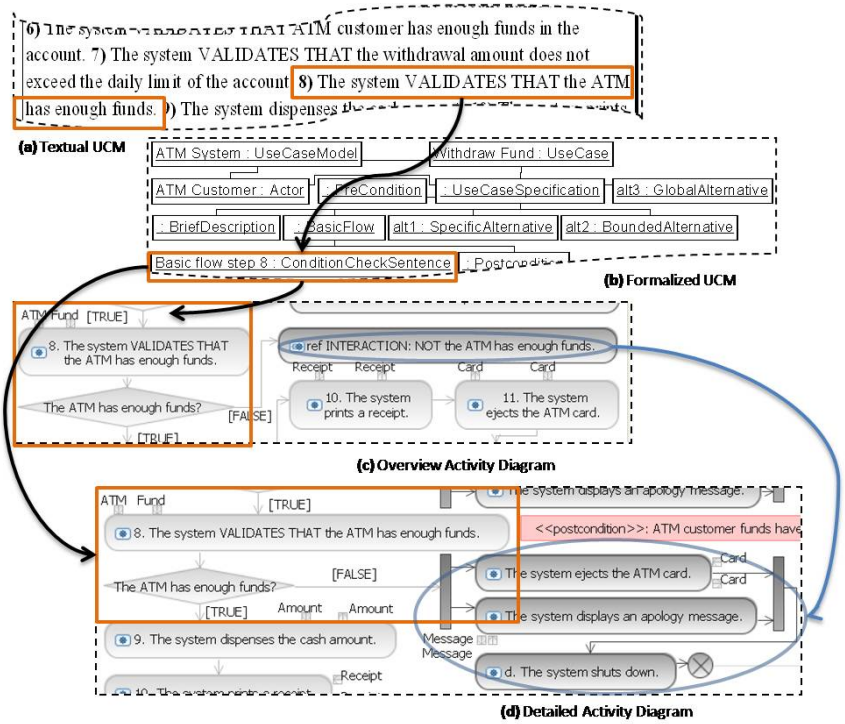

Fig. 1. Running example

fails) is specified in its entirety (circled set of nodes). Instead, in the overview activity diagram (Fig. 1 (c)), the alternative flow leads to a node labeled `ref INTERACTION …`, specifying that the alternative flow can be obtained from an interaction, specifically from a part of the sequence diagram aToucan generated for the use case.

Overview activity diagrams are similar to UML Interaction Overview Diagrams [19] in the sense that both use `CallBehaviorAction` referring to instances of `Interaction`s of sequence diagrams. However, UML Interaction Overview Diagrams only "focus on the overview of the flow of control where the nodes are `Interaction`s or `InteractionUse`s" [19] while our Overview Activity Diagram can contain other activity nodes such as instances of `CallOperationAction`.

During the transformations, two sets of traceability links are established: between the elements in the textual UCMod and the elements of the instance of UCMeta, and between these instances and the model elements of the UML analysis model. For instance, step 8 of the use case specification of Table 1 is linked to the `ConditionCheckSentence` instance highlighted in Fig. 1 (b), which is itself linked to the activity nodes labeled `8 (The System …)` in Fig. 1 (c) and (d). Notice that when necessary, direct traceability links between the textual UCMod and the analysis model can be easily derived from the transitive closure of these two sets of traceability links.

3.2 Transformation Rules

The transformation from an instance of UCMeta to activity diagrams involves 19 rules, summarized in Table 2. Subscripts on rule numbers (Column 1, Table 2)

indicate the type of the rule: "c" and "a" denote composite and atomic rules, respectively; a composite rule is decomposed whereas an atomic rule is not.

Rule 1 invokes rules 1.1-1.4 to generate an activity diagram for each use case. Rules 1.1-1.3 process three types of sentences: `SimpleSentence`, `ComplexSentence`, and `SpecialSentence`. Rule 1.4 processes the `GlobalAlternativeFlow`s of a use case. Rules 1.5 and 1.6 transform the precondition and the postcondition of a use case into instances of `Constraint` (a metaclass of the UML 2.0 metamodel) attached to the generated activity (precondition) and corresponding `FlowFinalNode` (postcondition). Atomic rules 1.2.1-1.2.4 are invoked by composite rule 1.2 to process four different types of complex sentences that lead to different control flows in the activity diagram (e.g., decision node in rule 1.2.1). Atomic rules 1.3.1-1.3.4 are invoked by rule 1.3 to process four different types of special sentences: to specify include and extend relations between use cases, to specify abort and resume. Rule 1.4 transforms a global alternative flow. Recall that a global alternative flow refers to any step in the reference flow (Section 2.1). For example, the global alternative flow of use case *Withdraw Fund* (Table 1) refers to every step of the basic flow; the ATM customer can cancel the transaction at any time of the execution of the use case. To model this in an activity diagram, we transform the flow into an instance of `AcceptEventAction`, `InterruptibleActivityRegion` and a set of actions corresponding to the steps of that flow. `AcceptEventAction` "is an action that waits for the occurrence of an event meeting a specified condition" [19] and `InterruptibleActivityRegion` (e.g., the basic flow of use case *Withdraw Fund*) is used to abort all flows in the region when an `AcceptEventAction` (e.g., ATM customer enters Cancel—the condition of the global alternative flow) occurs. Thanks to this modeling feature of UML 2.0, we can easily model global alternative flows.

Note that rules 1.2.1, 1.2.2, and 1.4 generate different sets of model elements for detailed and overview activity diagrams. For example, if an overview activity diagram is generated, composite rule 1.2.1 generates an instance of `CallBehaviorAction` to refer to the `Interaction` corresponding to the alternative flow of a condition check sentence; otherwise rule 1.2.1 invokes rules 1.1-1.3 to process the sentences of the alternative flow to generated actions, edges, etc, and create a detailed activity diagram.

Rule 2 invokes rules 2.1-2.3 to attach data flow information to an already generated activity diagram. These rules generate instances of either `InputPin` or `OutputPin` for each call operation action. These input and output pins correspond to entity classes that have been generated from the NPs contained in use case sentences when the class diagram of the system was generated [26]. For example, the basic flow step 8 of use case *Withdraw Fund* (Table 1) is a condition check sentence (Fig. 1 (a)). It is transformed into an action and a decision node by rule 1.2.1: (Fig. 1 (c) and (d)). Because the NP "enough funds"—i.e., the object of the condition (simple sentence) of the condition check sentence at step 8—has been transformed into class `Fund` when the class diagram was generated, we attach an instance of `InputPin` (`pin`) to the action corresponding to step 8 and type the pin with class `Fund`: `pin.type = Fund`, as show in Fig. 1 (c) and (d).

Table 2. Summary of transformation rules

Rule #	Description
1_c	Generate an activity diagram for a use case.
1.1_a	Generate an instance of `CallOperationAction` for each simple sentence.
1.2_c	Invoke rules 1.2.1-1.2.4 to process each complex sentence.
$1.2.1_c$	`ConditionCheckSentence`: Generate a `CallOperationAction` and a `DecisionNode`. Invoke rules 1.1-1.3 to handle the sentences contained in the alternative flow corresponding to the sentence (detailed activity diagram) or refer to the `Interaction` corresponding to the alt. flow (overview activity diagram).
$1.2.2_c$	`ConditionalSentence`: Generate a `DecisionNode`. Invoke rules 1.1-1.3 to process sentences contained in the sentence and its alt. flow (detailed activity diagram) or refer to the `Interaction` corresponding to the alt. flow (overview activity diagram) if such an alt. flow exists.
$1.2.3_c$	`ParallelSentence`: Generate a `ForkNode` and a `JoinNode`. Invoke rules 1.1-1.3 to process the concurrent sentences contained the parallel sentence.
$1.2.4_c$	`IterativeSentence`: Generate a `DecisionNode`. Invoke rules 1.1-1.3 to process sentences contained in the iterative sentence.
1.3_c	Invoke rules 1.3.1-1.3.4 to process each special sentence.
$1.3.1_a$	`IncludeSentence`: Generate a `CallBehaviorAction` that refers to the `Interaction` corresponding to the included use case.
$1.3.2_a$	`ExcludeSentence`: Generate a `CallBehaviorAction` that refers to the `Interaction` corresponding to the extending use case.
$1.3.3_a$	`AbortSentence`: Generate a `FlowFinalNode`.
$1.3.4_a$	`ResumeStepSentence`: Generate a `ControlFlow` edge back to the node corresponding to the step specified in the `ResumeStepSentence`.
1.4_c	`GlobalAlernativeFlow`: Generate an `AcceptEventAction` and `InterruptibleActivityRegion`. Invoke rules 1.1-1.3 to process the sentences of the alt. flow (detailed activity diagram) or refer to the `Interaction` corresponding to the alt. flow (overview activity diagram).
1.5_a	`Precondition`: Generate a `Constraint` as the precondition of the activity. The content of the constraint is the precondition of the use case.
1.6_a	`PostCondition`: Generate a `Constraint` for each flow final node as its postcondition. The content of each constraint corresponds to the postcondition of each flow of events of the UCMod.
2_c	Attach data flow information to an activity diagram.
2.1_a	`SimpleSentence` with transaction type `Initiation`, `ResponseToPrimaryActor` or `ResponseToSecondaryActor`: Generate an `OutputPin` for the `CallOperationAction` generated for the NPs of the sentence.
2.2_a	`SimpleSentence` with transaction type `InternalTransaction`: Generate an `InputPin` and an `OutputPin` for the `CallOperationAction` generated for the sentence (excluding "the system" and actors).
2.3_a	`ConditionCheckSentence`: Generate an `InputPin` for the `CallOperationAction` generated for the NPs of the sentence (excluding "the system" and actors).

The rationale for adding data flow to an activity diagram is the following. Steps of a UCS can be one of the following five types: 1) `Initiation`: the primary actor sends a request and data to the system; 2) `Validation`: the system validates a request and data; 3) `InternalTransaction`: the system alters its internal state (e.g., recording or modifying something); 4) `ResponseToPrimaryActor`: the system replies to the primary actor with a result; 5) `ResponseToSecondaryActor`: the system sends requests to a secondary actor. We generate data flow through input and output pins according to these definitions as follows: We generate output pins for actions in the activity diagram that correspond to use case steps of type `Initiation`, `ResponseToPrimaryActor`, and `ResponseToSecondaryActor` since these sentences either output data (`Initiation`) or send a result to actors (`ResponseToPrimaryActor` or `ResponseToSecondaryActor`); Since use case steps of type `InternalTransaction` specify that the system records or modifies data, we generate input and output pins for the actions corresponding to these sentences; Since condition check sentences are all

```
operation transform_rule_1.2.3(uc : ucm::UseCase, sen : uml::ParallelSentence, activity : uml::Activity,
                               partitions : Set<uml::ActivityPartition>, nodes : Sequence<uml::ActivityNode>,
                               edges : Sequence<uml::ActivityEdge>) : Void
    pre precondition is do
      sen.isInstanceOf(ucm::ParallelSentence) and activity != void
    end
    is do
      var message : traceability::Message   ---Instance of Message of the traceability metamodel
      message := trace_helper.createMessage("1.2.3", "ConditionalSentence -> ForkNode + JoinNode")
      var sens : Sequence<Sentence> init Sequence<Sentence>.new
      sens.addAll(sen.parallelActions)   ---The concurrent sentences contained in the parallel sentence
      var forkNode : uml::ForkNode init uml::ForkNode.new
      var joinNode : uml::JoinNode init uml::JoinNode.new
      var inEdge : uml::ActivityEdge    ---The incoming edge to the fork node
      inEdge := edges.last()   ---The source of the incoming edge is a previously generated activity node.
      inEdge.target := forkNode   ---The target of the incoming edge is the newly generated fork node.
      nodes.add(forkNode)   --- Store the fork node to the node collection.
      nodes.add(joinNode)   --- Store the join node to the node collection.
      activity.node.add(forkNode)  --- Connect the fork node to the activity.
      activity.node.add(joinNode)   --- Connect the join node to the activity.
      forkNode.inPartition.add(inEdge.source.inPartition.one)  ---Add the fork and join nodes to-
      joinNode.inPartition.add(inEdge.source.inPartition.one)  ---the partition as its previous node.
      sens.each{s |                                                    --- Process each concurrent sentence
          var outEdge : uml::ActivityEdge init uml::ControlFlow.new
          outEdge.source := forkNode   --- The outgoing edge from the fork node
          var parNodes : Sequence<ActivityNode> init Sequence<ActivityNode>.new
          var parEdges : Sequence<ActivityEdge> init Sequence<ActivityEdge>.new
          parEdges.add(outEdge)--- Maintain a branch for each concurrent sentence
          parNodes.add(forkNode)
          activity.edge.add(outEdge)
          if s.isInstanceOf(ucm::SimpleSentence) then
            transform_E1_1(uc, s.asType(ucm::SimpleSentence), activity, partitions, parNodes, parEdges)
          end
          if s.isInstanceOf(ComplexSentence) then
            transform_E1_2(uc, s.asType(ucm::ComplexSentence), activity, partitions, parNodes, parEdges)
          end
          if s.isInstanceOf(SpecialSentence) then
            transform_E1_3(uc, s.asType(ucm::SpecialSentence), activity, partitions, parNodes, parEdges)
          end
          var toJoinEdge : uml::ActivityEdge init uml::ControlFlow.new
          toJoinEdge.source := parNodes.last   --- The edge connecting to the join node
          toJoinEdge.target := joinNode
          parEdges.add(toJoinEdge)
          activity.edge.add(toJoinEdge)
      }
      var outgoingEdge : uml::ActivityEdge init uml::ControlFlow.new
      outgoingEdge.source := joinNode   --- The outgoing edge from the join node
      edges.add(outgoingEdge)
      activity.edge.add(outgoingEdge)
                                    --- Establish traceability links
      if forkNode != void then trace_helper.createTLink(sen, forkNode, message) end
      if joinNode != void then trace_helper.createTLink(sen, joinNode, message) end
end
```

Fig. 2. Transformation rule 1.2.3 in the Kermeta language [14]

of type `Validation` and they validate a request and data, input pins should be generated for the corresponding actions in the activity diagram. As suggested earlier, the pins are typed by the entity classes that compose the domain model (class diagram) automatically created from use case descriptions by aToucan [26], which are identified by analyzing sentences (e.g., subjects).

Notice that each rule is further specified by a precondition, although these preconditions are not shown in Table 2. As a simple example, the precondition of rule 1.2.3 specifies that the sentence being transformed into model elements is a parallel sentence (Section 2.1) and that an activity has been generated for the use case and is available to contain the elements being generated by the rule (**Fig. 2**).

As an example, consider composite rule 1.2.3 (**Fig. 2**) where a parallel sentence is transformed into a fork node and a join node and the concurrent sentences are transformed into actions between the fork and join nodes by invoking rules 1.1-1.3. Two traceability links are established between the fork and join nodes and the parallel complex sentence during this transformation. Traceability links are also established between each concurrent sentence (sentences connected by keyword MEANWHILE in the parallel sentence) and their corresponding actions when rules 1.1-1.3 are executed. The specification of the rule is shown as the Kermeta language [14], which is a metamodeling language similar to the Object Constraint Language (OCL) [20] in **Fig. 2**. Some of statements are annotated with *italic fonts* starting with "---".

3.3 Algorithm

Rules are structured in a hierarchy, as illustrated by the numbering in Table 2, which indicates that some rules have to be executed one after the other while others can be executed independently. This facilitates the modification, addition, and deletion of rules and also simplifies the algorithm to apply them. The invocation of each rule is determined by its precondition.

Fig. 3 presents the high level algorithm for transforming an instance of UCMeta into an activity diagram (an instance of the UML 2.0 activity metamodel). The algorithm is formalized using pseudocode where some variables are typed as model elements in UCMeta (prefixed with *ucm::*) and the UML activity diagram metamodel (prefixed with *uml::*), which are highlighted in `courier new` font.

The `Transform` function has one input parameter: the use case being transformed into an activity (instance of metaclass `UseCase` of UCMeta: `uc`) and outputs the generated activity (`activity`) for the use case. It starts by generating an activity, its initial node, and its partitions (lines 1-7): partitions (or swimlanes) are created for the system and each actor interacting with the use case. Second, the step sentences of the basic flow of the use case are transformed by invoking rules 1.1-1.3 (lines 8-12). Which rule is to invoke depends on the type of a sentence. Third, rule 1.4 is invoked to process the global alternative flows of the use case (line 13). Fourth, the final node, precondition, and postcondition of the activity are generated (lines 14-22). Fifth, rule 2 is invoked to generate data flow information for the already generated activity diagram (line 23). Finally, a traceability link is generated between the use case and the activity (line 24), as described next. Note that more specific links between the use case elements (e.g., sentence) and model elements of the activity are generated in each invoked rule (i.e., rules 1.1-1.4 and 2).

```
Algorithm   Transform (uc) : activity
Input       uc : ucm::UseCase --- The use case being transformed into an activity
Output      activity : uml::Activity --- The activity transformed from the use case
Declare     basicFlowSteps : Sequence(ucm::Sentence) --- The basic flow steps of the use case
            altFlows : Set(ucm::AlternativeFlow) --- The alternative flows of the use case
            initialNode : uml::InitialNode --- The initial node of the activity
            partitions : Set(uml::ActivityPartition) --- The partitions of the activity
            finalNode : uml::ActivityFinalNode --- The final node of the activity
            precondition : uml::Constraint --- The precondition of the activity
            postcondition : uml::Constraint --- The postcondition of the main branch of the activity
            tlink : traceability::Trace --- The traceability link between the use case and the activity
            msg : traceability::Message--- The tlink message describing the transformation.
Begin
 1. activity ← uml::Activity.new
 2. activity.name ← uc.name
 3. initinalNode ← uml::InitialNode.new
 4. activity.node.add(initialNode)
 5. partitions.add(CreatePartitionForSystem(uc))
 6. partitions.add(CreatePasrtitionForEachActor(uc))
 7. activity.partition.addAll(partitions)
 8. basicFlowSteps.each { s : ucm::Sentence |
 9.    if (s.isInstanceOf(ucm::SimpleSentence)  then invoke_rule_1.1(uc, s, activity)  end
10.    if (s.isInstanceOf(ucm::ComplexSentence) then invoke_rule_1.2(uc, s, activity)  end
11.    if (s.isInstanceOf(ucm::SpecialSentence) then invoke_rule_1.3(uc, s, activity)  end
12. }
13. invoke_rule_1.4(uc, altFlows.select{a|a.isInstanceOf(ucm::GlobalAlternativeFlow)}, activity)
14. finalNode ← uml::ActivityFinalNode.new
15  activity.node.add(finalNode)
16. precondition ← uml::Constraint.new
17. precondition.name ← uc.specification.preCondition.content
18  activity.ownedRule.add(precondition)
19. postcondition ← uml::Constraint.new
20. postcondition.name ← uc.specification.flows.
21.             select{f|f.isInstanceOf(ucm::BasicFlow)}.one.postCondition.content
22. activity.ownedRule.add(postcondition)
23. invoke_rule_2(uc, activity)
24. tlink ← createTLink(uc, activity, message)
25. return activity
End
```

Fig. 3. Transformation algorithm

3.4 Traceability

We establish two sets of traceability links during the transformation from a textual UCMod to activity diagrams: from the UCMod to the instance of UCMeta; from the UCMeta instance to the automatically generated activity diagrams. If necessary, direct traceability links from the textual UCMod to the activity diagrams can be derived from these two sets.

Traceability links from UCMod to UCMeta link the fields of the use case template used to document textual UCSs to instances of the corresponding metaclasses in UCMeta. For example, field *Brief Description* of the use case template is linked to an instance of metaclass `BriefDescription` of UCMeta. A sentence in the brief description is then linked to an instance of metaclass `Sentence` of UCMeta. For example, as shown in Fig. 1 (a) and (b), the basic flow step 8 of use case *Withdraw Fund* is transformed into `Basic flow step 8 : ConditionCheckSentence` of the intermediate model while a traceability link is established between these two elements.

We believe that it is not cost effective to establish links at a finer granularity (e.g., between elements of sentences and UCMeta metaclass instances).

Regarding the second set of traceability links, UCMeta metaclass instances are linked to corresponding model elements in the UML activity metamodel based on our transformation rules. For example, we establish two traceability links between a condition check sentence (e.g., `Basic flow step 8 : ConditionCheckSentence` of the UCMeta instance as shown in Fig. 1 (b)) and its corresponding action (an instance of `CallOperationAction`, e.g., action `8. The system VALIDATES THAT the ATM has enough fund` as shown in Fig. 1 (c)) and decision node (e.g., `The ATM has enough fund` as shown in Fig. 1 (c)) generated during the transformation when rule 1.2.1 is invoked, respectively.

4 Automation

Our approach has been implemented as part of aToucan [26]. aToucan aims to automatically transform requirements given as a UCMod in RUCM into a UML analysis model including a class diagram, and a set of sequence and activity diagrams. It relies on a number of existing technologies. aToucan is built as an Eclipse plug-in, using the Eclipse development platform. UCMeta is implemented as an Ecore model, using Eclipse EMF [8], which generates code as Eclipse plug-ins. The Stanford Parser [22] is used as a NL parser in aToucan. It is written in Java and generates a syntactic parse tree for a sentence and the sentence's grammatical dependencies (e.g., *subject*, *direct object*). The generation of the UML analysis model relies on Kermeta [14]. It is a metamodeling language, also built on top of the Eclipse platform and EMF. The target UML analysis model is instantiated using the Eclipse UML2 project, which is an EMF-Based implementation of the UML 2 standard.

The architecture of aToucan is easy to extend and can accommodate certain types of changes. Transformation rules for generating different types of diagrams are structured into different packages to facilitate their modifications and extensions. Thanks to the generation of an Eclipse UML2 analysis model, generated UML models can be imported and visualized by many open source and commercial tools. Similarly, though UCSs are currently provided as text files, a specific package to import UCSs will allow integration with open source and commercial requirement management tools. More details on the design of aToucan can be found in [26].

We adapted the traceability model proposed in the traceability component (fr.irisa.triskell.traceability.model) of Kermeta [14] to establish traceability links. Details of the traceability model is discussed in [26].

5 Case studies

In this section, we discuss how we validated our approach (Section 5.1) and also compare our approach with three commercial tools (Section 0).

5.1 Validation procedure and summary of results

We used five different software system descriptions (18 use cases altogether) to assess our approach. They are from different sources: three are from textbooks and

two were created by Masters students. Since the UCSs of these systems come from different sources, they were re-written by applying RUCM.

The goal of our validation was two-fold: (1) To assess whether our transformation rules are complete: does it accommodate all UCSs in our case studies? (2) To determine whether our transformation rules lead to activity diagrams that are syntactically and semantically correct. Syntactic correctness means that a generated activity diagram conforms to the UML 2.0 activity diagram notation. Semantic correctness means that a generated activity diagram correctly represents its UCS; all the steps described in the flows of events of the UCS are correctly transformed by following the transformation rules and no redundant model elements are generated. In order to check correctness and completeness, the validation procedure is as follows.

1. Given a UCMod in RUMC as input, aToucan automatically generates an activity diagram for each UCS of the UCMod.
2. For each UCS, we check whether each step of the flows of events and the precondition and postconditions have been properly transformed.
3. We check whether each generated activity diagram is syntactically and semantically correct.
4. We check whether the data flow information (input and output pins) attached to each activity diagram is properly generated.

Following the above procedure, for all 18 use cases, we achieved 100% completeness and correctness with aToucan, and 100% of the traceability links were also correctly established. Regarding the completeness and correctness of data flow information attached to each activity diagram, aToucan was not able to generate input and output pins for some actions. First, transformation rules 2.1-2.3 (used to generate data flow information) rely on package `SentenceStructure` of UCMeta. Recall that a NL parser is used in our approach to parse each textual sentence and the parsing result is transformed into instances of model elements (e.g., `Object`) of package `SentenceStructure` (Section 2.2). The NL parser has limitations and cannot always produce a correct result. Therefore the instances of the model elements of package `SentenceStructure` do not always correctly correspond to their textual sentences. Second, each generated pin is typed to a class of the class diagram; however the automatically generated class diagram is not 100% correct and complete (see [26] for details), again partly because of limitations of the NL parser. As a result, it is possible that there is no matching class found for an element of a sentence (such as an object—recall rule 2 in Section 3.2) and therefore no pin is generated. Besides, whether data flow information can be deemed correct also depends on what it is used for. For example, if it is used to automatically generate test cases, manual refinement of the automatically generated data flow information is absolutely required. Therefore, here, we don't evaluate the correctness of generated data flow information but its completeness, measured by the ratio of occurrences of missing pins over the total number of instances of `CallOperationAction` in an activity diagram. Table 3 summarizes the completeness of data flow information of every use case of all the five case study systems. Results show that the average completeness of data flow information across all the five case study systems is 85%.

Table 3. Completeness of data flow information

Case studies	Use cases	Total # of `CallOperationAction`	Occurrences of missing pins
ATM	Withdraw Fund	22	1
	Query Account	8	1
	Transfer Fund	17	2
	Validate PIN	16	3
Elevator	Select Destination	13	2
	Request Elevator	14	2
CPD	Order Part	8	1
	Insufficient Stock	4	1
	Create Vendor Order	6	2
	Create Preventive Order	5	1
	Create Customer Order	5	0
	Complete Pending Order	6	1
VS	Rent Video	13	2
	Reserve Video	12	1
	Return Video	9	2
	Check Database	15	2
	Video Overdue	9	1
ARENA	Announce Tournament	24	4
Total		**206**	**29**
%		**29/206 = 85%**	

5.2 Comparison with three commercial tools

Visual Paradigm [23], Ravenflow [21], and CaseComplete [6] are commercial tools that can automatically transform requirements into UML activity diagrams. We tested them by using the use case of Table 1 since it contains three different types of alternative flows, concurrency sentences, and validation sentences. Various features of UCSs are therefore considered and this use case can be considered complete in terms of UCS and generated activity diagram features. The UCS was rewritten according to the format requirements of each tool. The details of the re-written UCSs and automatically generated activity diagrams are provided in Appendix A for reference. In the rest of the section, we summarize their main differences.

1. Visual Paradigm and CaseComplete can transform the flows of events of a use case into an activity diagram. Each flow of events needs to be structured using a simple use case template (basic flow and its extensions). Ravenflow does not require a use case template, but a set of writing guidelines are proposed (not enforced by the tool though) to guide users to write sentences that can be correctly parsed by the tool. For example, "if...then.... Otherwise,..." is suggested to write a conditional sentence. A "!' at the end of a sentence indicates the termination of a flow. Since Ravenflow does not require UCSs be structured, alternative flows may be very hard to describe in unstructured sentences. aToucan is based on RUCM (a use case template and a set of restriction rules), which have been experimentally evaluated to be easy to apply [25]. The benefits of using RUCM to facilitate the automated generation of activity diagrams was discussed in Section 2.1.
2. None of the three commercial tools can generate forks and joins because concurrency sentences are not recognized. Our approach is based on RUCM,

which specifies the keyword MEANWHILE (Section 2.1) to help users specify concurrency sentences. Therefore, aToucan can generate a fork, a join, and a set of parallel sentences between the fork and the join (Section 3.2) for each concurrency sentence. Visual Paradigm and CaseComplete do not support swimlanes. Both our approach and Ravenflow support swimlanes—one swimlane per actor and one swimlane for the system—but have different mechanisms to identify actors. Ravenflow relies on Natural Language Processing (NLP) techniques to identify possible actors to generate corresponding swimlanes, which means that the tool might falsely identify actors. Recall that Ravenflow does not have a use case template to structure use case steps. However, our tool is based on RUCM, and primary and secondary actors of each use case are clearly specified in each UCS. Also thanks to RUCM, aToucan can also automatically transform global alternative flows (Section 3.2). It is very hard (if not impossible) to find an alternative way to specify global alternative flows in the requirements format required/enforced by the three commercial tools.

3. None of the three commercial tools can support include and extend use case relationships because they can only transform a single use case instead of a use case model (UCMod). aToucan takes a UCMod as input and use case relationships are naturally supported.
4. Visual Paradigm and CaseComplete cannot generate any data flow information since they do not use any NLP technique. Ravenflow can generate data flow information but to a quite limited extent: it generates data flow only when the data is manipulated by two swimlanes, as indicated in the writing guidelines provided along with the tool. This means that Ravenflow cannot derive data flow information from sentences with transaction types `InternalTransaction` and `Validation`. aToucan does not have such a limitation.

6 Related Work and Comparison

We conducted a systematic literature review [27] on transformations of textual requirements into analysis models, including class, sequence, and activity diagrams. The review identified 20 primary studies (16 approaches) based on a carefully designed paper selection procedure in scientific journals and conferences from 1996 to 2008 and Software Engineering textbooks. The method proposed here is based on the results of this review, with a particular focus on automatically deriving activity diagrams from UCMods. There also exists several literature works recently published that were therefore not included in our systematic literature review. In this section, we evaluate our approach by comparing it with these existing literature works and also three existing commercial tools: Visual Paradigm for UML [23], Ravenflow [21], and CaseComplete [6]. We define a set of evaluation criteria for comparison, which are in part from the system review we conducted [27]:

1) **Requirements**: We need to know the requirements format (e.g., formalized use cases) required by a specific approach so that we can assess how difficult it is to document requirements.
2) **NLP**: We need to be aware of whether or not any NLP techniques are applied. We can then assess whether or not certain features (e.g., automatically derive

swimlanes) can be supported by the approach.

3) **Automation**: This criterion evaluates whether a transformation is automated, automatable, semi-automated, or manual. An approach is automated if it has been fully implemented. If a transformation algorithm is proposed in a paper, then we assess whether we deem the description to be sufficient to implement it, and if this is the case, the transformation approach is deemed automatable. In some cases, a transformation is semi-automated because user interventions are required. Last, some approaches are entirely manual.
4) **Traceability**: We check whether traceability links between requirements and analysis model elements are established when a transformation is performed.
5) **Objective**: The original objective of each approach can help us understand their limitations and motivate our work.
6) **Activity diagram**: We evaluate the activity diagrams that each approach is able to derive from requirements with respect to the following four aspects: 1) their types (standard, extended, or non-standard notation), 2) important model elements that are expected to be generated (e.g., swimlanes), 3) whether include and extend relationships of use cases are supported, and 4) whether data flow information can be generated. Activity diagrams conforming to the UML specification [19] are standard activity diagrams; extended activity diagrams are those based on a profile of the UML specification; non-standard activity diagrams do not conform to the UML specification.

The evaluation results are summarized in Table 4 and Table 5. The first columns of these two tables show the approaches we evaluated. The first four rows are the approaches proposed in existing research works; the following three rows are the selected commercial tools; the last row is our approach: aToucan. The rest of the columns are arranged according to the evaluation criteria.

Table 4 Evaluation summary (part I)

Approach	Requirements	NLP	Automation	Trace ability	Objective
[12]	Formalized UCs	No	Automated	No	Visualize use cases and facilitate test generation
[13]	Unstructured requirements	Yes	Automatable	No	Complement analysis models
[9]	Unstructured requirements	Yes	Semi-automated	No	Complement analysis models
[16]	Exceptional UCs	--	Manually	No	Formalize UCs to reduce ambiguities
[23]	Flows of events of UCs	No	Automated	Yes	Visualize flows of events of UCs
[21]	Restricted sequential steps	Yes	Automated	Yes	Visualize textual sequential steps
[6]	Flows of events of UCs	No	Automated	No	Visualize flows of events of UCs
aToucan	**RUCM models**	**Yes**	**Automated**	**Yes**	**All of the above**

Table 5 Evaluation summary (part II)

Approach	Activity Diagram						
	Type	Swimlane	Fork and join	Decision node	Global alternative flows	Include or extend	Data flow
[12]	Standard	Yes	No	Yes	No	No	No
[13]	Extended	No	No	Yes	No	--	No
[9]	Standard	No	Yes	No	No	--	No
[16]	Extended	--	--	--	--	--	--
[23]	Standard	No	No	Yes	No	No	No
[21]	Standard	Yes	No	Yes	No	--	Yes
[6]	Non-standard	No	No	No	No	No	No
aToucan	**Standard**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

As shown in Column 2, Table 4, one approach [12] formalizes use cases as instances of a metamodel, similarly to aToucan. However, the metamodel instance has to be manually provided by the user directly, instead of being transformed automatically from another (more simple) representation (RUCM), thereby leading to substantial user effort. Two approaches ([13] and [9]) take unstructured requirements (plain text) as inputs to derive activity diagrams. Both are not fully automated. An approach is proposed in [16] to manually transform exceptional use cases into extended activity diagrams. Special stereotypes (e.g., `<<failure>>` and `<<handler>>`) are introduced to specify exceptional handling concepts. Visual Paradigm [23] and CaseComplete [6] can automatically transform flows of events of a use case into an activity diagram. Both tools require a similar and simple use case template to structure flows of events. Ravenflow [21] can automatically visualize a set of sequential and textual steps into an activity diagram and no structured format (e.g., template) is needed to document these steps. Ravenflow however suggests users follow a set of writing principles, some of which are very similar to the restriction rules of RUCM used in aToucan. Because Ravenflow does not rely on a use case template, it becomes very difficult to specify alternative flows in a use case and their interactions with the basic flow.

As shown in Column 3, Table 4, except for one manual approach, three existing approaches rely on NLP techniques. The approach proposed in [12] does not apply any NLP techniques because it requires formalized use cases as its inputs. Visual Paradigm [23] does not rely on any NLP technique; therefore it cannot automatically generate swimlanes, forks and joins, for instance. Four existing approaches are automated (Column 4) and only two commercial tools have traceability capability (Column 5).

The objectives of the existing approaches are different, as shown in Column 6, Table 4. The approach proposed in [12] aims to visualize use cases and therefore automated test generation can be facilitated. Visualizing use cases or their scenarios for the purpose of better understanding and analyzing them is a common practice [2, 10] and the idea of activity diagram-based test generation is also promoted in [7, 17]. Both the approaches proposed in [13] and [9] can generate analysis models including class and activity diagrams. Generated activity diagrams, as part of the generated analysis models, model dynamic behavior of a system. The approach proposed in [16] however simply uses activity diagrams as a means to formalize textual use cases. All three commercial tools visualize either flows of events of use cases (i.e., [6, 23]) or sequential textual steps (i.e., [21]) in activity diagrams for the purpose of helping users to construct and understand requirements. Our approach however applies to any of these objectives.

The approaches proposed in [13, 16] cannot generate standard UML activity diagrams (Column 2, Table 5). Mustafiz et al. [16] propose an approach to manually transform exceptional use case (with elements that allow the modeling of system behavior in exceptional situations) into activity diagrams extended by specific stereotypes. Ilieva and Ormandjieva [13] propose an automatable approach to transform requirements into extended activity diagrams—activity diagrams integrated with the concepts of actors, business rules, and messages. As shown in Columns 3-6, except for the manual approach, swimlanes are only supported by two approaches: one of them [12] requires formalized use cases as input and the other [21] relies on

NLP techniques to automatically identify swimlanes (similarly to aToucan); only one approach [9] supports forks and joins but it is semi-automated; decision nodes are supported by most of the non-manual approaches; Global alternative flows are not supported by any of the existing approaches. The approaches that are not manual and take use cases as inputs, do not support include and extend relationships of use cases. Ravenflow is the only existing approach that can generate data flow information (similarly to aToucan), and we have discussed differences in Section 0.

To compare with these existing approaches, our approach can automatically transform each use case of a UCMod into two types of standard UML 2.0 activity diagrams while fully supporting traceability. Additionally, as part of the functionality of aToucan, automatically generated activity diagrams are naturally consistent with other UML analysis model diagrams such as sequence diagrams and horizontal traceability (across different diagrams) can then be supported. Swimlanes, decision nodes, forks and joins, include and extend relationships, and data flow information are all supported by our approach. Besides, thanks to RUCM, our approach can also transform global alternative flows.

7 Conclusion

Providing automated support to derive UML analysis models, including class, sequence and activity diagrams from use case requirements is an important step of model-driven development. Even if such models end up being incomplete and an initial step to converge towards satisfactory models, the potential benefits are substantial. However this step has not received enough attention in large part because requirements (e.g., Use Case Specifications—UCSs) are essentially textual documents and tend to be unstructured. Therefore their automated analysis is difficult to achieve.

In this paper, we propose an approach, supported by the aToucan tool [26], to automatically generate activity diagrams from a Use Case Model documented by using a use case modeling approach (RUCM) specifically designed to facilitate this automated transition. Additionally, traceability links can be generated between requirements and activity diagrams while transformations are performed. Automatically generated activity diagrams could be used, for example, to visualize use cases for the purpose of better understanding and analyzing them to model behavioral aspects of systems as part of analysis models, or to facilitate test case generation.

Two types of activity diagrams (overview and detailed activity diagrams) can be generated from use cases while traceability links can be established between the use cases and their corresponding activity diagrams. An overview activity diagram for a use case shows only the main flow of the use case and models its alternative flows by referring to parts of the sequence diagram aToucan generates for the use case. By doing so, the automatically generated activity diagram can be greatly simplified, which is important especially when the use case is very complex (e.g., many alternative flows). A detailed activity diagram, however provides a complete view of all the flows (main and alternative) of a use case. Our approach can also automatically and selectively (according to users' desire) attach data flow information into already generated activity diagrams. This information can be used to measure design complexity and facilitate data flow-based testing.

Five case studies have been performed using the aToucan tool. As expected, results show that our approach can generate higher quality activity diagrams than alternative approaches (including three commercial tools) based on a number of evaluation criteria. These criteria relate to automation, traceability, the completeness of activity diagrams, and the ease of writing requirements.

8 References

1. IEEE Std. 830-1998, IEEE Standard for Software Requirement Specification (1998)
2. Berenbach, B., Inc, S.C.R., Princeton, N.J.: The evaluation of large, complex UML analysis and design models. ICSE (2004)
3. Bittner, K., Spence, I.: Use Case Modeling. Addison-Wesley Boston (2002)
4. Brown, E.K., Miller, J.E.: Syntax: a linguistic introduction to sentence structure. Routledge (1992)
5. Bruegge, B., Dutoit, A.H.: Object-Oriented Software Engineering Using UML, Patterns, and Java. Prentice Hall (2009)
6. CaseComplete: http://www.casecomplete.com/
7. Chen, T.Y., Tang, S.F., Poon, P.L., Tse, T.H.: Identification of categories and choices in activity diagrams. QSIC 2005. Citeseer (2005) 55-63
8. Eclipse Foundation: Eclipse Modeling Framework.
9. Fliedl, G., Kop, C., Mayr, H.C., Salbrechter, A., Vöhringer, J., Weber, G., Winkler, C.: Deriving static and dynamic concepts from software requirements using sophisticated tagging. Data Knowl. Eng. **61** (2007) 433-448
10. Fowler, M.: UML distilled: a brief guide to the standard object modeling language. Addison-Wesley (2003)
11. Gomaa, H.: Designing Concurrent, Distributed, and Real-Time Applications with UML. Addison-Wesley (2000)
12. Gutiérrez, J.J., Clémentine, N., Escalona, M.J., Mejías, M., Ramos, I.M.: Visualization of Use Cases through Automatically Generated Activity Diagrams. MODELS2008. Springer
13. Ilieva, M.G., Ormandjieva, O.: Models Derived from Automatically Analyzed Textual User Requirements. Soft. Eng. Research, Management and Applications (2006)
14. Kermeta: Kermeta metaprogramming environment. Triskell team
15. Kruchten, P.: The Rational Unified Process: An Introduction. Addison-Wesley (2003)
16. Mustafiz, S., Kienzle, J., Vangheluwe, H.: Model transformation of dependability-focused requirements models. ICSE Workshop on Modeling in Software Engineering (2009)
17. Nebut, C., Fleurey, F., Le Traon, Y., Jezequel, J.M.: Automatic test generation: A use case driven approach. IEEE TSE **32** (2006) 140-155
18. Olsen, G.K., Oldevik, J.: Scenarios of traceability in model to text transformations. ECMDA-FA, Haifa, Israel (2007)
19. OMG: UML 2.2 Superstructure Specification.
20. OMG: OCL 2.0 Specification. Object Management Group (2003)
21. RAVENFLOW: http://www.ravenflow.com/
22. The Stanford Natural Language Processing Group. The Stanford Parser version 1.6
23. Visual Paradigm for UML: http://www.visual-paradigm.com/product/vpuml/
24. Waheed, T., Iqbal, M.Z.Z., Malik, Z.I.: Data Flow Analysis of UML Action Semantics for Executable Models. ECMDA-FA (2008)
25. Yue, T., Briand, L.C., Labiche, Y.: A Use Case Modeling Approach to Facilitate the Transition Towards Analysis Models: Concepts and Empirical Evaluation. MODELS2009. Springer
26. Yue, T., Briand, L.C., Labiche, Y.: Automatically Deriving a UML Analysis Model from a Use Case Model. Carleton University (2009)

27. Yue, T., Briand, L.C., Labiche, Y.: A Systematic Review of Transformation Methodologies between User Requirements and Analysis Models. Carleton University (2009)

Appendix A UCSs and automatically generated activity diagrams

In this appendix, we provide the re-written UCSs and automatically generated activity diagrams of aToucan and the three commercial tools compared with aToucan: Visual Paradigm, CaseComplete, and Ravenflow.

A.1 aToucan

The re-written UCS by applying RUCM is provided in Table 1. The automatically generated overview and detailed activity diagrams are presented in Fig. 5 and Fig. 4, respectively. Data flow information for the detailed activity diagram is given in **Table 6**-not directly attached to it to avoid overcrowding the diagram.

As shown in **Table 6**, no input pins were generated for action 12 and d. The sentence corresponding to action d is "The system shuts down." Therefore the action has no input pin. Because NL parser cannot correctly parse the sentence corresponding to action 12: "The system displays Welcome message". So this is an error. An input pin should be generated for the action.

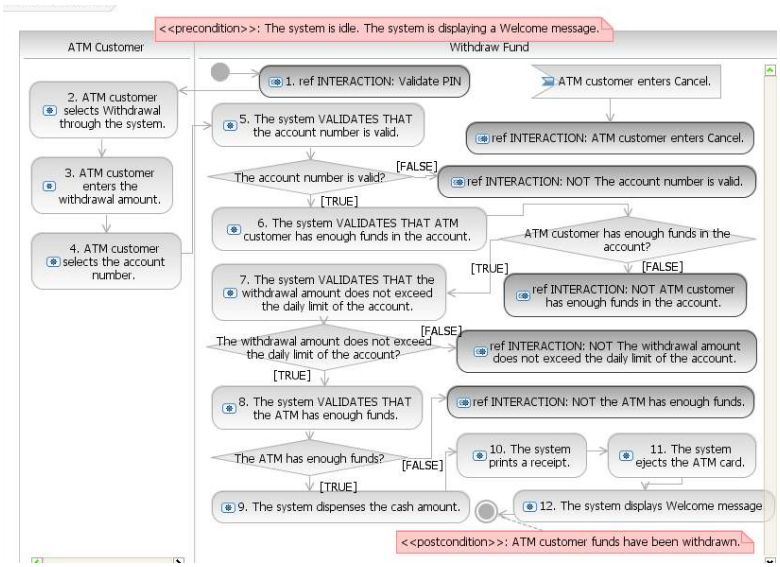

Fig. 4. Overview Activity Diagram – aToucan

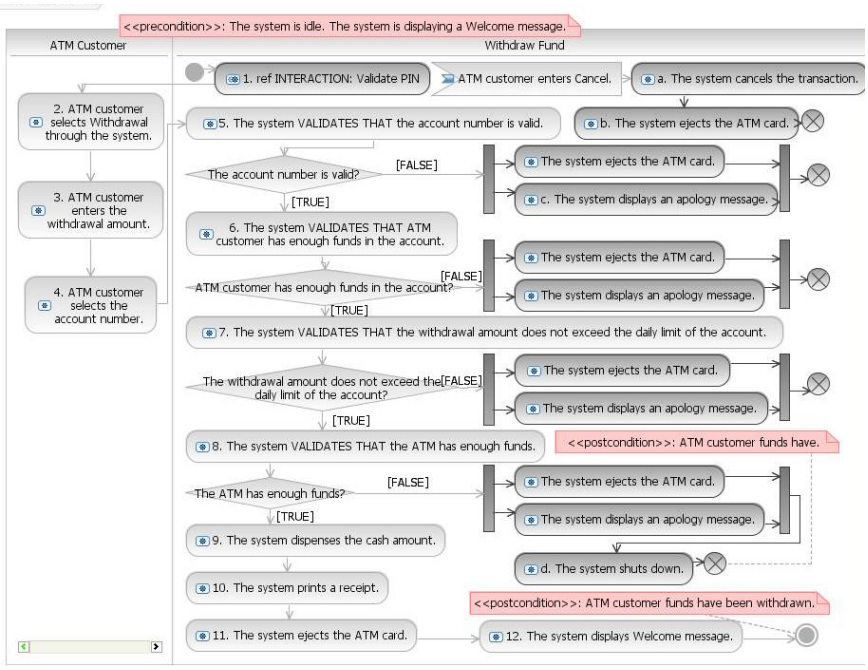

Fig. 5. Detailed Activity Diagram – aToucan

Table 6. Data flow informatin on detail activity diagram

Action	Typed class(es) of pins
2 (out)	Withdrawal
3 (out)	Amount
4 (out)	Account Number
5 (in)	Account number
6 (in)	Customer, Fund, Account
7 (in)	Amount
8 (in)	ATM, Fund
9 (in)	Amount
10 (in)	Receipt
11 (in)	Card
12 (in)	--
a (in)	Transaction
b (in)	Card
c (in)	Message
d (in)	--

A.2 Visual Paradigm

The re-written UCS according to the format enforced by Visual Paradigm is provided in Fig. 6. The automatically generated activity diagram is presented in Fig. 7.

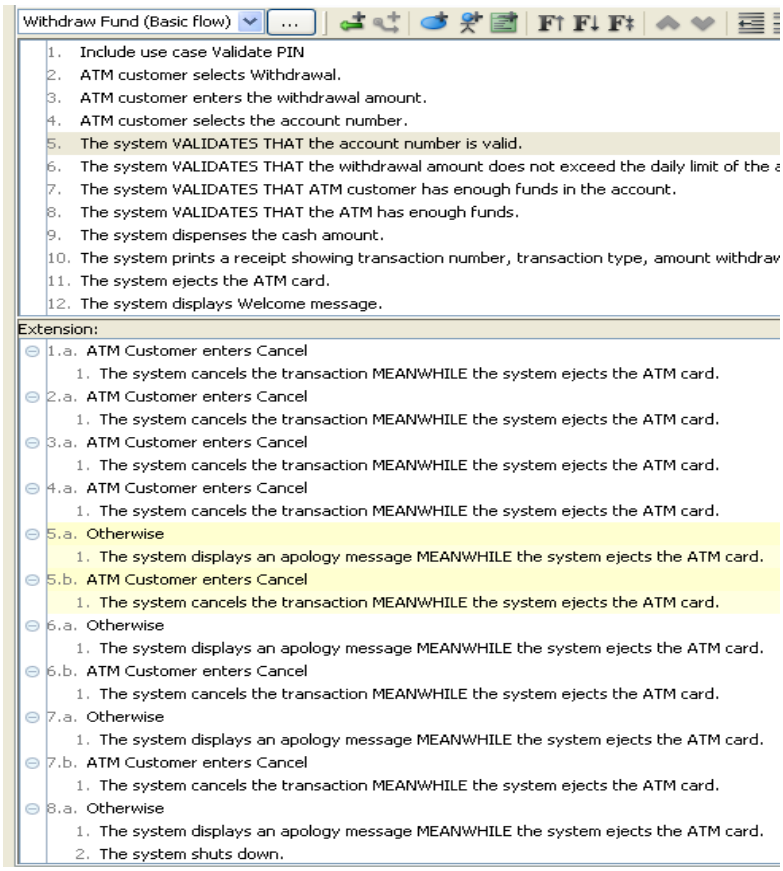

Fig. 6. Re-written UCS - Visual Paradigm (partial)

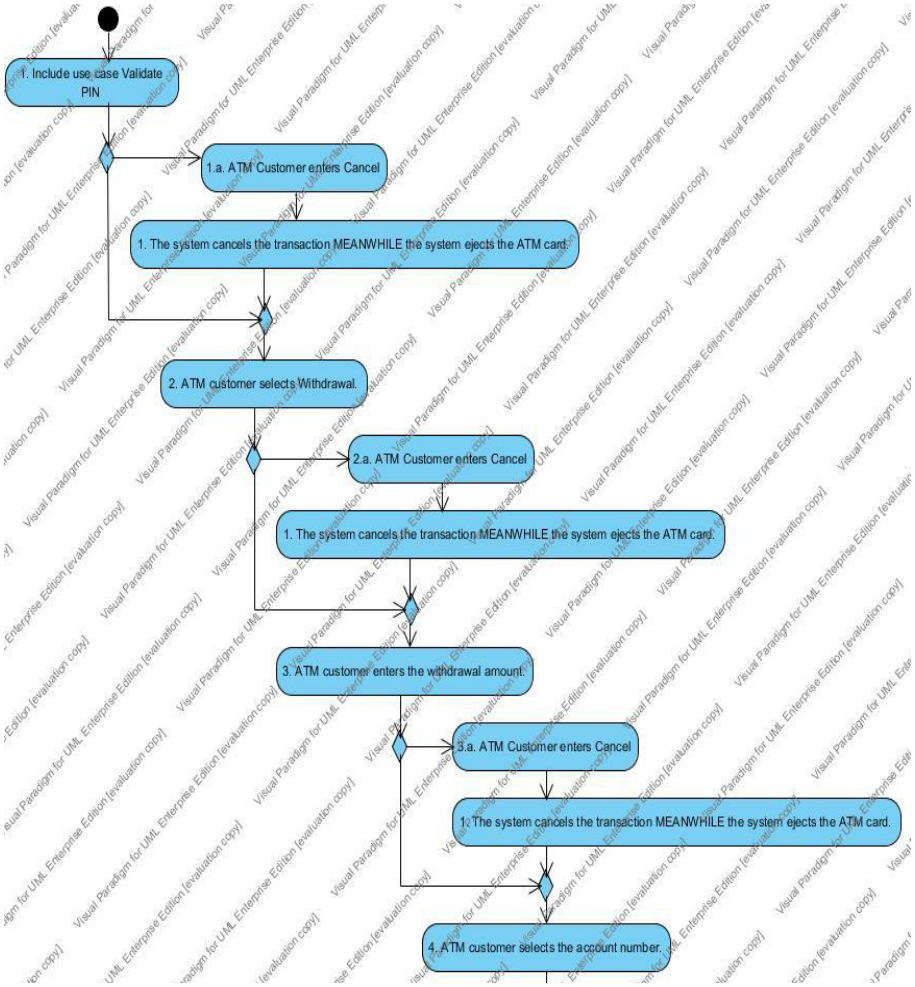

Fig. 7. Activity Diagram - Visual Paradigm (partial)

A.3 Ravenflow

The re-written UCS according to the format enforced by Ravenflow is provided in Fig. 9. The automatically generated activity diagram is presented in Fig. 8. Notice that there is data flow information was generated for the activity diagram.

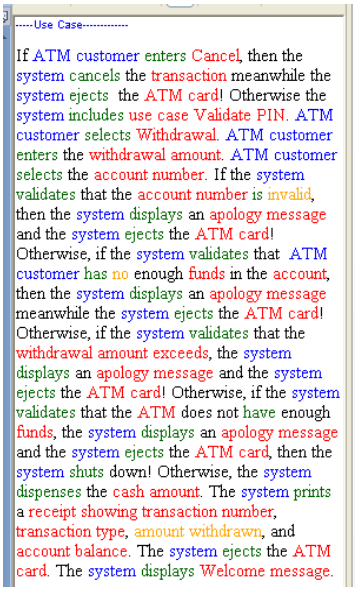

Fig. 9. Re-written UCS - Ravenflow

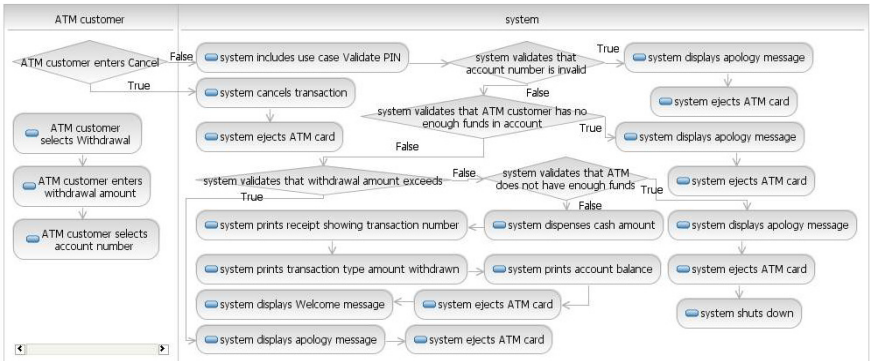

Fig. 8. Activity Diagram - Ravenflow (for better layout, the activity diagram is exported to RSA and visualized by it)

A.4 CaseComplete

The re-written UCS according to the format enforced by CaseComplete is provided in Fig. 10. The automatically generated activity diagram is presented in Fig. 11.

Steps | Prose | Expand | Show Testing Procedure

Main Success Scenario:

1. Include use case Validate PIN
2. ATM customer selects Withdrawal.
3. ATM customer enters the withdrawal amount.
4. ATM customer selects the account number.
5. The system VALIDATES THAT the account number is valid.
6. The system VALIDATES THAT ATM customer has enough funds in the account.
7. The system VALIDATES THAT the withdrawal amount does not exceed the daily limit of the account.
8. The system VALIDATES THAT the ATM has enough funds.
9. The system dispenses the cash amount.
10. The system prints a receipt showing transaction number, transaction type, amount withdrawn, and account balance.
11. The system ejects the ATM card.
12. The system displays Welcome message.

Extensions:

1.a. If ATM customer enters Cancel then
   1. The system cancels the transaction MEANWHILE the system ejects the ATM card.
2.a. If ATM customer enters Cancel then
   1. The system cancels the transaction MEANWHILE the system ejects the ATM card.
3.a. If ATM customer enters Cancel then
   1. The system cancels the transaction MEANWHILE the system ejects the ATM card.
4.a. If ATM customer enters Cancel then
   1. The system cancels the transaction MEANWHILE the system ejects the ATM card.
5.a. else
   1. The system displays an apology message MEANWHILE the system ejects the ATM card.
5.b. If ATM customer enters Cancel then
   1. The system cancels the transaction MEANWHILE the system ejects the ATM card.
6.a. else
   1. The system displays an apology message MEANWHILE the system ejects the ATM card.
6.b. If ATM customer enters Cancel then
   1. The system cancels the transaction MEANWHILE the system ejects the ATM card.
7.a. else
   1. The system displays an apology message MEANWHILE the system ejects the ATM card.
7.b. If ATM customer enters Cancel then
   1. The system cancels the transaction MEANWHILE the system ejects the ATM card.
8.a. else
   1. The system displays an apology message MEANWHILE the system ejects the ATM card.
   2. The system shuts down.
8.b. If ATM customer enters Cancel then
   1. The system cancels the transaction MEANWHILE the system ejects the ATM card.
9.a. If ATM customer enters Cancel then
   1. The system cancels the transaction MEANWHILE the system ejects the ATM card.

Fig. 10. Re-written UCS - CaseComplete (partial)

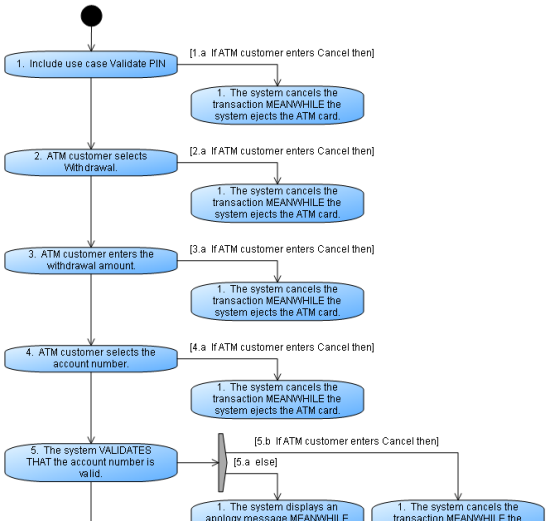

Fig. 11. Activity Diagram - CaseComplete (partial)