Information Extraction from High-level Activity Diagrams to Support Development Tasks

Martin Beckmann, Thomas Karbe and Andreas Vogelsang
Technische Universität Berlin, Ernst-Reuter-Platz 7, 10587 Berlin, Germany

Keywords: UML2 Activity Diagrams, Information Extraction, Activity Semantics.

Abstract: As the complexity of systems continues to increase, the use of model-driven development approaches becomes more widely applied. One of our industry partners (Daimler AG) uses UML activity diagrams as the first step in the development of vehicle functions, mainly for the purpose of communication and overview. However, the contained information is also valuable for further development tasks. In this paper, we present an automated approach to extract information from these high-level activities. We put a focus on aspects of activities such as propositional logic relations, sequences of actions, and differentiability of execution paths. The extracted parts are needed for the compilation of requirements and the creation of test cases. Also, this approach supports stakeholders unfamiliar with the notations of activities as implicit information is made explicit and hence more accessible. For this purpose, we provide a formalism for the kind of activities our industry partner uses. Based on that formalism, we define properties that express the contained sequences and execution paths. Furthermore, the formalism is used to derive the underlying propositional logic relations. We show how the approach is applied to eliminate hundreds of existing quality issues in an existing requirements document.

1 INTRODUCTION

Complex software systems, which, for example, can be found in distributed embedded systems, require model-based and system-oriented development approaches (Broy, 2006). Also, using graphical models for specification manages complexity and improves reusability and analytical capabilities (Vogelsang et al., 2014). One of our industry partners (Daimler AG) uses UML activity diagrams as a first step for developing a new function of a vehicle system. The activities describe the function's activation and deactivation in terms of triggers and conditions that need to be checked and fulfilled before a function is activated. By this, the activity diagrams provide an early overview of the desired function behavior.

Although the main purpose of the diagrams is to be a means of communication and to ease the overall understanding, the contained information is also a valuable input for following development tasks such as the elaboration and documentation of detailed requirements (Drusinsky, 2008) or the derivation of test cases (Kundu and Samanta, 2009). Yet, different tasks have different information needs and may benefit from making explicit specific information contained in the activity diagrams. We aim at supporting downstream development tasks by extracting and preparing the relevant information from the activity diagrams. This extraction is additionally helpful for stakeholders unfamiliar with the notations of activity diagrams (Arlow and Neustadt, 2004) because it makes information contained in the activity diagrams more accessible (Maiden et al., 2005).

In this paper, we focus (1) on the transformation of activity diagrams to textual specifications by exploiting information on logical activation expressions and (2) on supporting the derivation of test cases by exploiting information on minimal execution sequences. This paper makes the following contributions:

- We define a simplified representation of UML activities based on graphs. For this simplified representation, we define an algorithm that computes minimal execution sequences within the activity and a second algorithm that computes an activation expression for a function.
- We use the information about minimal execution sequences to derive test cases from the activities.
- We show how we use the activation expressions to derive textual requirements specifications from the activities.
- For both applications, we report on our experiences gained at our industrial partner.

Beckmann, M., Karbe, T. and Vogelsang, A.
Information Extraction from High-level Activity Diagrams to Support Development Tasks.
DOI: 10.5220/0006605504380445
In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2018)*, pages 438-445
ISBN: 978-989-758-283-7

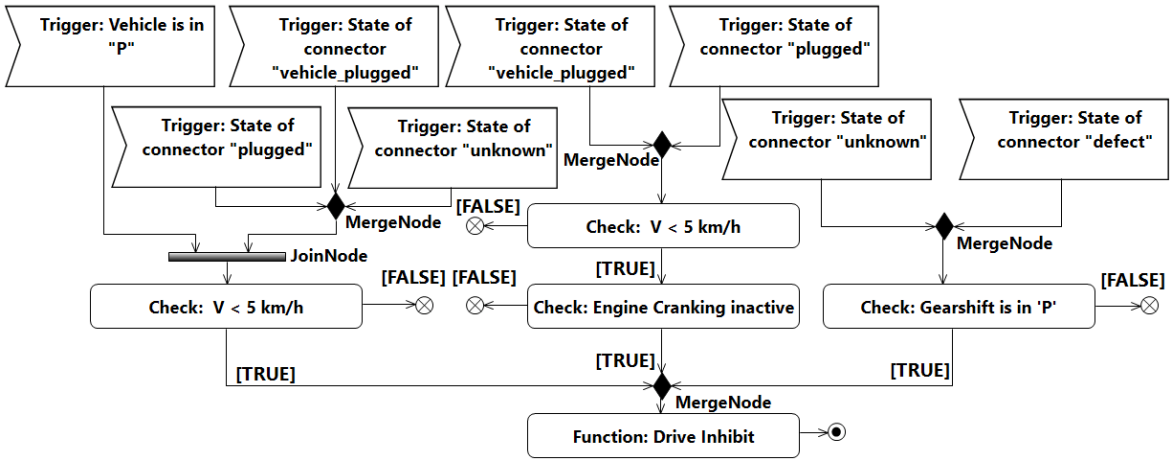


Figure 1: Activity diagram of the function *Drive Inhibit*.

2 BACKGROUND

Our industry partner uses UML2 activity diagrams to specify functions of a system. These activity diagrams are the first step of the development of a new system function. They are used to get an early overview of the desired function behavior. Although the main focus of the activity is to be a means of communication and to make the understanding easier, it already contains a number of information that can be used in the following development phases such as the elicitation and documentation of requirements and the derivation of test cases.

Figure 1 shows the activity diagram of the function *Drive Inhibit*. The actual behavior of the activated function is described in the *Action* node labeled with *Drive Inhibit* (bottom of the diagram). The function's activation is described by a combination of triggers and checks for conditions. For triggers, the *AcceptEventAction* element is used. The checks are modeled as *Action* elements. If the condition of a check is not fulfilled, the flow ends (*FlowFinal*). As a consequence, a check acts as an implicit AND. The triggers and checks are connected by *ControlNodes* such as *JoinNodes* and *MergeNodes*. *JoinNodes* act as synchronization points and can be interpreted as AND operators in terms of propositional logic. *MergeNodes* represent OR operators. Once the actual functionality of the function is executed, *ActivityFinal* elements designate the end of an activity.

Relevant information for our industry partner concerns (amongst others): (minimal) execution paths, propositional logic relations and sequential or independent executability of actions.

Execution paths are of interest for testing and to facilitate the planning of the system. They are the basis to derive test cases that ensure that the function is in fact activated, when certain action are executed. The execution paths also provide information about the sequences of execution of *actions*. This can be combined with the mapping of the involved actions to the components of the system (this mapping is not part of the activity). As a result, it is possible to make statements on the dependencies between the involved components. This knowledge is applied during the planning of the development of the system.

Propositional logic relations are needed to derive requirements that describe the correct behavior of the system as well as the test cases that validate these requirements.

3 RELATED WORK

As this paper introduces a formalism for a certain kind of activities, it is related to work about formal semantics of UML2 activities. The UML2 Specification describes *Activities* as Petri net like graphs (Object Management Group (OMG), 2015, p. 283), but does not provide formal semantics. Therefore a number of formal semantics have been proposed, i.a. (Störrle, 2004). While most approaches try to cover the capabilities to a full extent, it is considered useful to express activities in simpler constructs (Lano, 2009). We use this idea and present a formalism solely devoted to derive information about certain aspects in activities.

Graphical models are the basis for a number of approaches that derive different software engineering artifacts from the models. Amongst others, they are used to automatically generate source code (Usman and Nadeem, 2009) and test cases (Kundu and

Samanta, 2009). Using graphical models and especially UML to generate textual requirements or parts of requirements documents has already been covered by a number of research papers (Nicolás and Toval, 2009). Specifically activities as a source for requirements have already been addressed by Drusinsky (Drusinsky, 2008), however, only for UML-1. Additionally, we take into account propositional logic relations, execution paths, and allow for queries on actions about independent executions.

In contrast to the mentioned approaches, our approach focuses on extracting certain aspects of activities and does not restrict itself on a single application.

4 EXTRACTING INFORMATION FROM ACTIVITIES

For the purpose of this paper, we aim at extracting specific information from activities to facilitate downstream development tasks. More specifically, we want to extract the following information:

Independent Actions. Independent actions within an activity can be executed without any interrelations. This information is useful for the planning of the development. Actions are executed by components of the system. From the independence of actions follows that there is no flow of information between the components and hence development can progress without considering the component executing an independent action.

Minimal Execution Paths. A minimal execution path for a node within an activity is a set of actions that need to be executed before the node can be executed. These paths contain all actions that are logically required for a token to reach an action. Superfluous actions occurring in parallel are not part of the minimal execution path. This information is useful for the creation of test cases. The test cases verify that a function is executed due to or in spite of certain conditions. Using minimal paths ensures that only conditions are tested that influence the examined executed path. This leads to a minimal set of tests, which are necessary to confirm the behavior of a function.

Activation Expressions. An activation expression for a node within an activity is a propositional expression that reflects the logical relations between the preceding actions of the node. The activation expression abstracts from any order of execution and can be used to derive textual specifications corresponding to the activities.

In the following, we present how these information can be extracted from the activities.

4.1 Activity Graphs

To extract the information on independent actions and minimal execution paths, we introduce activity graphs as a simplified representation of the activities. Activity graphs focus on expressing whether certain actions are independent of one another or whether they have to be executed sequentially. We transform an activity to an activity graph by mapping the actions of an activity to nodes of a graph. We assume that implicit connections in the activity are made explicit and that *ExecutableNodes* only appear once in the activity. Beckmann et al. have proposed an approach that we use to remove redundant occurrences of *ExecutableNodes* within an activity (Beckmann et al., 2017a). There may be cycles in the activity.

Each node in the activity graph has a label containing the text of the corresponding *Action* of the activity. They also have one of the following types: Trigger, Check, Function, Merge, Decision, Join, Fork. Moreover, each node has a set of successors.

Definition 1. ***Activity Graph***
Given a non-empty set of nodes V, an activity graph *T is defined as*

$$T \stackrel{\text{def}}{=} (V, suc_T, type_T, label_T)$$

where

1. *$suc_T : V \to \mathcal{P}(V)$ is the successor function for T, where $suc_T(v)$ denotes the set of all successor nodes of $v \in V$,*
2. *$type_T : V \to \{Trigger, Check, Function, Merge, Decision, Join, Fork, End\}$ assigns a type to every node, and*
3. *$label_T : V \to \Sigma^*$ assigns a label to every node.*

Definition 2. ***Direct Predecessors***
Given an activity graph T, the set of direct predecessors *of a node $v \in V$ is defined as*

$$dpred_T(v) \stackrel{\text{def}}{=} \{w \mid v \in suc_T(w)\}$$

In the activity the direct predecessor is the source node of any incoming edge. There might be more than one direct predecessor to one node. Since we assume that all connections were made explicit and there are no redundant elements, multiple direct predecessors occur only for *JoinNodes* and *MergeNodes*.

Definition 3. ***Execution Sequence***
Given an activity graph T,

1. *A list of nodes $s = \langle v_1, \ldots, v_n \rangle$ with $v_1, \ldots, v_n \in V$ is called an* execution sequence*, and v_i is called the* i-th execution step of *s.*
2. *An execution step v_i is a* sequence-predecessor *of another execution step v_j (denoted by $v_i <_s v_j$) if $i < j$.*

3. *The set of all execution sequences of T is denoted by S.*

Considering Figure 1 one possible execution sequence might be *Trigger: State of connector "unknown"*, *Check: Gearshift is in 'P'*, *Function: Drive Inhibit*.

Definition 4. ***Prefix***
Given an activity graph T and an execution sequence $s = \langle v_1, \ldots, v_n \rangle$. For any k with $1 \leq k \leq n$ the k-prefix (or just prefix) of s *is defined by*

$$s_{(k)} \stackrel{\text{def}}{=} \langle v_1, \ldots, v_k \rangle.$$

Definition 5. ***Node Count***
Given an activity graph T and an execution sequence $s = \langle v_1, \ldots, v_n \rangle$. For any node $v \in V$, the node count of v in s *is a function $\#_v(s) : S \to \mathbb{N}$ and describes the number of appearances of v in s.*
Example:

- $\#_a(\langle a,b,c,a,d,e,a,d \rangle) = 3$,
- $\#_e(\langle a,b,c,a,d,e,a,d \rangle) = 1$,
- $\#_f(\langle a,b,c,a,d,e,a,d \rangle) = 0$.

Definition 6. ***Valid Execution Sequence***
Given an activity graph T and an execution sequence $s = \langle v_1, \ldots, v_n \rangle$,

1. *An execution step v_i of a sequence s is* valid *(denoted by $s \vdash_T v_i$) if and only if one of the following cases is true:*
   (a) $dpred_T(v_i) = \emptyset$,
   (b) $dpred_T(v_i) \neq \emptyset \wedge type(v_i) \neq Merge \wedge \forall w \in dpred_T(v_i).\#_w(s_{(v_i)}) \geq \#_{v_i}(s_{(v_i)})$
   Explanation: A join node is valid when each of its predecessors appears at least as often as the join node itself in the prefix before it. Check, Function, Fork, Decision, and End nodes are similar, but have only one predecessor. The formula is the same for them.
   (c) $dpred_T(v_i) \neq \emptyset \wedge type(v_i) = Merge \wedge \#_{v_i}(s_{(v_i)}) <= \sum_{w \in dpred_T(v_i)} \#_w(s_{(v_i)})$
   Explanation: A merge node is valid when all its predecessors together appear at least as often as the merge node itself in the prefix before it.
2. *An execution sequence s is* valid *(denoted by $\vdash_T s$), when all its execution steps are valid.*
3. *The set of all valid execution sequences for T is denoted by S_T.*

Definition 7. ***Predecessor***
Given an activity graph T, a node $v_i \in V$ is a predecessor *of another node $v_j \in V$ (denoted by $v_i <_T v_j$) if v_i is a sequence predecessor of v_j in every valid sequence of T.*

$$v_i <_T v_j \Leftrightarrow \forall s \in S_T . v_i <_s v_j$$

This definition is used to find dependencies between actions. In case a node is predecessor of another node, the predecessor has to be executed first. This also tells us that there is an interaction between nodes.

Definition 8. ***Independent Nodes / Parallel Executable***
Given an activity graph T, two nodes $v_i, v_j \in V$ are independent *(denoted by $v_i \parallel_T v_j$) if they are not predecessors of each other:*

$$v_i \parallel_T v_j \Leftrightarrow v_i \not<_T v_j \wedge v_j \not<_T v_i$$

In contrast to the predecessor relation, two independent nodes can be executed without any interrelations between the involved actions. An example in Figure 1 are the checks *$V < 5$ km/h* and *Gearshift is in 'P'*.

Definition 9. ***Minimal Execution***
Given an activity graph T and a node $v \in V$. A minimal execution sequence *$s_{min,T}(v) = \langle v_1, \ldots, v_n \rangle$ is a valid execution sequence that ends in v and for which no i exists for which $1 \leq i < n$ and $\langle v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n \rangle$ is valid.*
Note, that $v_n = v$ because the sequence ends in v.
Explanation: An execution sequence is minimal when no step can be cut out of the sequence.

Every path to the specified node that does not contain unnecessary actions for the activation, is a minimal execution. Since *MergeNodes* might have multiple predecessors, there can be more than one minimal execution. The action *Check: $V < 5$ km/h* after the *JoinNode* in Figure 1 has three minimal executions. Each path consists of one of the three triggers connected by the *MergeNode*, the *MergeNode* itself, the *JoinNode* and the action *Trigger: Vehicle is in 'P'*.

Definition 10. ***Concatenation of Execution Sequences***
Given two execution sequences of disjoint nodes $s_1 = \langle v_1, \ldots, v_n \rangle$ and $s_2 = \langle w_1, \ldots, w_m \rangle$. The concatenated execution sequence *$s_1 \circ s_2$ is defined as*

$$s_1 \circ s_2 \stackrel{\text{def}}{=} \langle v_1, \ldots, v_n, w_1, \ldots, w_m \rangle$$

The algorithm to compute a minimal execution is shown in Algorithm 1. The algorithm works recursively through the graph. In a each step the necessary minimal executions are concatenated to the current node. Which executions are necessary depends on the type of the node. In case a node is neither a *JoinNode* nor a *MergeNode*, the minimal execution is the concatenation of the minimal execution of its direct predecessor and itself. For a *JoinNode*, all previous minimal executions are needed. For a *MergeNode*, any of the predecessor can be used. Hence,

Algorithm 1: Recursively computing a minimal execution.

Input: Activity Graph T, Node $v \in V$

function MINEX(v)
  if $dpred_T(v) = \emptyset$ **then**
    return $\{v\}$
  else if $type(v_i) \notin \{Merge, Join\}$ **and** $dpred_T(v) = \{w\}$ **then**
    return MINEX(w) $\circ \langle v \rangle$
  else if $type_T(v) = Join$ **and** $dpred_T(v) = \{w_1, \ldots, w_n\}$ **then**
    return MINEX(w_1) $\circ \cdots \circ$ MINEX(w_n) $\circ \langle v \rangle$
  Note, this step is not deterministic, since depending on the order of concatenation there are multiple options. Only one choice is needed.
  else if $type_T(v) = Merge$ **and** $w \in dpred_T(v)$ (any predecessor) **then**
    return MINEX(w) $\circ \langle v \rangle$
  Note that this step is not deterministic, since multiple predecessors can exist. Any choice would be correct.
  end if
end function

there are multiple minimal executions. The algorithm terminates, if there are no predecessors or if a cycle is detected. Executions containing cycles are discarded, because they cannot be minimal executions.

4.2 Activation Expressions

Actions that are predecessors of other actions in an activity diagram can also be interpreted as logical facts that need to be fulfilled before an action can be executed. Activation expressions focus on these logical relations between actions. These relations can be represented by a propositional logic expression tree. The algorithm to construct the activation expression for a node in an activity graph is displayed in Algorithm 2.

The algorithm requires the node for which the activation expression shall be computed as input. In our case, we are especially interested in action nodes that represent function executions. Some of the activities of our industry partner contain more than one function. In that case, multiple trees have to be created since each function has different triggers and checks, and thus, the activation expression is also different. As a second input, the algorithm requires a node of the tree that is to be created. The input is required since the algorithm works recursively. When the algorithm is called for the first time, a *start* node is used

Algorithm 2: Recursively computing an expression tree.

Input: Node $v_{Act} \in V_{Act}$, Node $v_{Tree} \in V_{Tree}$

function CREATEEXPTREE(v_{Act}, v_{Tree})
  if $dpred_{Act}(v_{Act}) = \emptyset$ **then**
    $suc_{Tree}(v_{Tree}) = suc_{Tree}(v_{Tree}) \cup v_{Act}$
  else if $dpred_{Act}(v_{Act}) \neq \emptyset$ **and** $type_{Act}(v_{Act}) \in \{Trigger, Check, Function\}$ **then**
    $v_{Treenext} \stackrel{\text{def}}{=} createNode(AND)$
    $suc_{Tree}(v_{Tree}) = suc_{Tree}(v_{Tree}) \cup v_{Treenext}$
    $suc_{Tree}(v_{Treenext}) = suc_{Tree}(v_{Treenext}) \cup v_{Act}$
    $v_{Act} \stackrel{\text{def}}{=} v \in dpred_{Act}(v_{Act})$
    $v_{Tree} \stackrel{\text{def}}{=} v_{Treenext}$
    createExpTree(v_{Act}, v_{Tree})
  else if $dpred_{Act}(v_{Act}) \neq \emptyset$ **and** $type_{Act}(v_{Act}) = Join$ **then**
    $v_{Treeand} \stackrel{\text{def}}{=} createNode(AND)$
    $suc_{Tree}(v_{Tree}) = suc_{Tree}(v_{Tree}) \cup v_{Treeand}$
    for all $v_{Act\,in}$ of $dpred_{Act}(v_{Act})$ **do**
      $v_{Act} \stackrel{\text{def}}{=} v_{Act\,in}$
      $v_{Tree} \stackrel{\text{def}}{=} v_{Treeand}$
      createExpTree(v_{Act}, v_{Tree})
    end for
  else if $dpred_{Act}(v_{Act}) \neq \emptyset$ **and** $type_{Act}(v_{Act}) = Merge$ **then**
    $v_{Treeor} \stackrel{\text{def}}{=} createNode(OR)$
    $suc_{Tree}(v_{Tree}) = suc_{Tree}(v_{Tree}) \cup v_{Treeor}$
    for all $v_{Act\,in}$ of $dpred_{Act}(v_{Act})$ **do**
      $v_{Act} \stackrel{\text{def}}{=} v_{Act\,in}$
      $v_{Tree} \stackrel{\text{def}}{=} v_{Treeor}$
      createExpTree(v_{Act}, v_{Tree})
    end for
  else if $dpred_{Act}(v_{Act}) \neq \emptyset$ and $type_{Act}(v_{Act}) \in \{Fork, Decision\}$ **then**
    $v_{Act} \stackrel{\text{def}}{=} v \in dpred_{Act}(v_{Act})$
    createExpTree(v_{Act}, v_{Tree})
  end if
end function

as the root node of the tree. What the algorithm basically does, is to traverse the activity graph backwards. It starts from the node that represents the function that has to be activated. From there the predecessors are analyzed until the triggers of the function or nodes without any predecessors are reached. As a consequence, the algorithm terminates as long as there is no cycle in any of the execution sequences. This can be automatically ensured beforehand by checking for

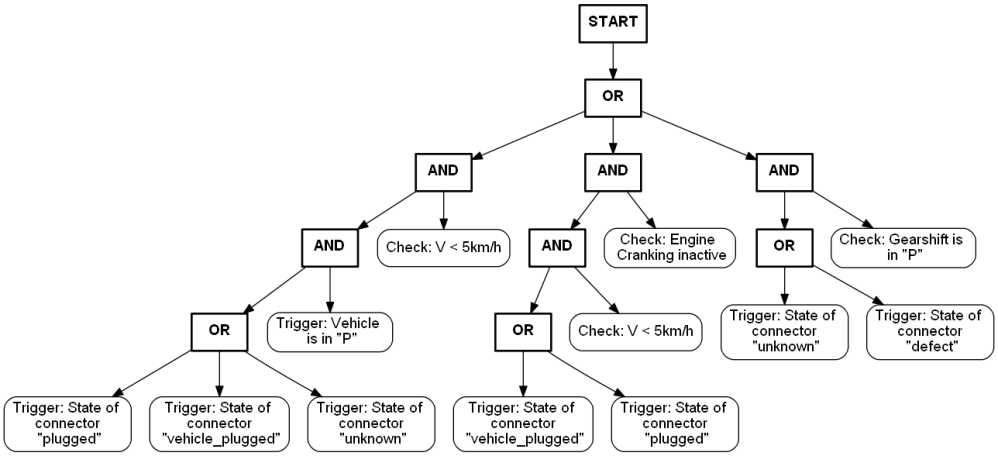
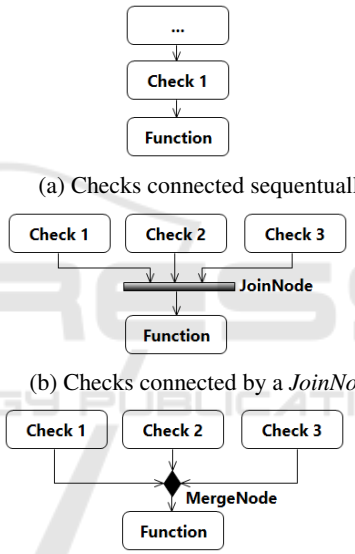


Figure 2: Expression Tree of the function *Drive Inhibit*

cycles. Also, for the activities our industry partner uses, a detected cycle can be ignored. This is possible, since the repeated execution of actions does not have any influence on the function activation. If the actions in the cycle were executed once, the flow of tokens also continues outside the cycle. Further repetitions do not effect that flow.

In each step of the traversal, the type of the current activity node is examined. Depending on the type, the nodes that are appended to the tree, differ. In case the examined node is an *Action* (e.g., a check), it means it has to be executed successfully for the flow to continue. This is depicted in Figure 3a. The traversal of the activity starts from the function. Before the function can be executed, a check must be fulfilled. Besides, there might be other nodes before the check. As this represents an *AND* connection, an *AND* node is added to the tree, and the found check is added to that new *AND* node. The resulting tree is shown in Figure 4a. The following recursive call uses the added *AND* node as the tree node input. The following activity nodes are then added to the *AND*. If a *JoinNode* or *MergeNode* is found in the activity, an *AND* or *OR* node is appended to the tree respectively. In contrast to a single action, these *ControlNodes* might have more than one predecessor. Exemplar activities for the *JoinNode* and the *MergeNode* are shown in Figure 3b and Figure 3c respectively. All predecessors are added to these tree nodes. The corresponding expression trees to the activities in Figure 3b and Figure 3c are shown in Figure 4b and Figure 4c. There is no negation operator, since there are no actions that undo events and hence stop the flow of tokens.

The corresponding expression tree to the activity in Figure 1 is displayed in Figure 2. The tree nodes that represent the operators (START, AND, OR) are displayed in square boxes, while the actual *ActivityNodes* are displayed as oval boxes. As a result of the algorithm, the *ExecutableNodes* of the original activity are the leaves of the tree.



(a) Checks connected sequentually

(b) Checks connected by a *JoinNode*

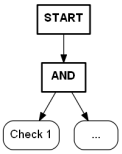
(c) Checks connected by a *MergeNode*

Figure 3: Different situations in activities.

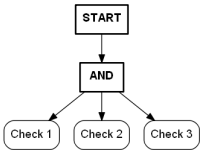
5 APPLICATIONS AND LIMITATIONS

5.1 Applications

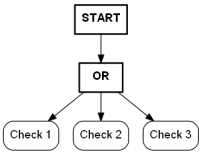
We used the introduced algorithms and definitions to support different development tasks in practice.



(a) Corresponding expression tree to situation in Figure 3a



(b) Corresponding expression tree to situation in Figure 3b



(c) Corresponding expression tree to situation in Figure 3c

Figure 4: Resulting expression trees.

5.1.1 Transformation of Activity Diagrams to Textual Specifications

In industry, graphical models such as activity diagrams cannot be used as the sole means of specification. Textual requirements complementing the activities are needed because of legal considerations (Sikora et al., 2012; Maiden et al., 2005) and to provide a systematic display of derived information (Weber and Weisbrod, 2002). Recent studies have found that practitioners prefer textual requirements specifications that are structured according to the different logical cases that may lead to a specific event (Beckmann and Vogelsang, 2017). Therefore, we used the structure of the activation expression tree to generate complementing textual requirements specifications for 36 activity diagrams of our industry partner. That way, we eliminated hundreds of different existing quality issues of a previous version.

Figure 5 shows the textual requirements derived from the activity of Figure 1. The excerpt shows explicitly the propositional logic relations by using the operators *AND* and *OR*. All elements connected by the same operator were placed one level below. This kind of structure equals the structure of the activation expression tree. Hence, we could directly map the result of the underlying propositional logic to the document structure. Prior studies have shown that manual creation and maintenance of textual requirements from diagrams is error-prone and labor-intensive (Beckmann et al., 2017b). An automatic model-to-text transformation based on our algorithm prevents quality issues and may save time.

Text	Level	Type
Drive Inhibit	2	Function
OR	3	-
AND	4	-
Vehicle is in "P"	5	Trigger
OR	5	-
State of connector "plugged"	6	Trigger
State of connector "vehicle_plugged"	6	Trigger
State of connector "unknown"	6	Trigger
V < 5 km/h	5	Check
AND	4	-
OR	5	-
State of connector "plugged"	6	Trigger
State of connector "vehicle_plugged"	6	Trigger
V < 5 km/h	5	Check
Engine Cranking inactive	5	Check
AND	4	-
OR	5	-
State of connector "defect"	6	Trigger
State of connector "unknown"	6	Trigger
Gearshift is in "P"	5	Check

Figure 5: Derived Textual Requirements.

5.1.2 Derivation of Test Cases

The approach was applied to recreate parts of already existing test cases for the displayed function *Drive Inhibit* in an automatic manner as a proof-of-concept. These parts encompass the name of the test case as well as templates for the test steps that must be performed to conduct the test case. The test steps must be added manually as they are not part of the activity diagram. The test cases ensure that the function is activated due to certain occurring events and fulfilled conditions. The necessary states and circumstances were directly derived from the identified minimal executions. The minimal executions of an action in the activity contain all necessary actions (i.e., events) that must appear and conditions that must be fulfilled to start an execution. As a result, test cases that describe in which states a function is activated can be directly derived since a minimal execution only contains these necessary conditions. Consequently every minimal execution is used to derive one test case. The created test cases can therefore assure that the function is in fact executed under the intended circumstances. Hence, using this approach ensures that all necessary conditions for executions are tested. For example in Figure 1 this leads to the creation of seven test cases. Three test cases originate from the three triggers connected to the trigger *Vehicle is in "P"* by a *JoinNode*. Two test cases are created for each pair of the two triggers connected by the *MergeNodes*.

In addition, non-minimal sequences can also be useful. The execution of superfluous actions makes sure the function is still activated when the necessary actions were executed. Also, it can be checked whether the function is activated, although necessary conditions are not met. In that case necessary actions are not executed.

5.2 Limitations

We focused on the capability of activities to describe sequences, parallelism, execution paths and propositional logic relations. Still, activities can be used in other ways to describe other aspects of behavior. Consequently, it is not possible to foresee every application. Thus, it is necessary to restrict oneself to certain aspects. While the extracted information can be used for multiple purposes, there are use cases that require different aspects our approach does not yet cover. One of these aspects are asynchronous events that are potentially used to abort the execution of an activity. These were not part of our work, since our industry partner does not use them.

Also, this work focuses on the activity diagrams of our industry partner. These activities only incorporate a subset of elements in activities. Still, this kind of description is quite common to describe functions (Firesmith, 2004). As a result, the approach is not generally applicable but we think that it provides a benefit for that kind of graphical descriptions.

6 CONCLUSION AND OUTLOOK

In this paper we presented an approach to extract implicitly contained information from high-level activities to support downstream development tasks. For this purpose we introduce activity graphs as a simplified, yet formal, representation of activity diagrams, which can be used to make statements about sequences and execution paths of activities. We show in detail how this can be used to derive textual requirements, which both improves the quality of the resulting requirements document and saves effort in its creation. Also, the creation of test cases was performed as a proof-of-concept for one function.

Furthermore, it is planned to use the extracted information for impact analysis. By combining the activities with the mapping of the actions to the components, dependencies between components are made more easily accessible. This knowledge will be used to derive visual architectural views of the whole system, which in turn shall facilitate release planning.

As the approach is restricted to a subset of elements, the approach is not generally applicable to all activities. Incorporating all elements (such as guards) of activities into the approach is an open issue. Also, there are further aspects of activities that are needed during the development of systems we did not yet consider. Which aspects need to be included and what artifacts they might be used for is also worth investigating.

REFERENCES

Arlow, J. and Neustadt, I. (2004). *Enterprise patterns and MDA: Building better software with archetype patterns and UML*. Addison-Wesley Professional.

Beckmann, M., Michalke, V., Vogelsang, A., and Schlutter, A. (2017a). Removal of Redundant Elements within UML Activity Diagrams. In *Conference on Model Driven Engineering Languages and Systems*.

Beckmann, M. and Vogelsang, A. (2017). What is a Good Textual Representation of Activity Diagrams in Requirements Documents? In *Model-Driven Requirements Engineering Workshop*.

Beckmann, M., Vogelsang, A., and Reuter, C. (2017b). A Case Study on a Specification Approach using Activity Diagrams in Requirements Documents. In *International Requirements Engineering Conference*.

Broy, M. (2006). Challenges in automotive software engineering. In *Proceedings of the 28th international conference on Software engineering*.

Drusinsky, D. (2008). From UML activity diagrams to specification requirements. In *International Conference on System of Systems Engineering*.

Firesmith, D. (2004). Generating Complete, Unambiguous, and Verifiable Requirements from Stories, Scenarios, and Use Cases. *Journal of Object Technology*.

Kundu, D. and Samanta, D. (2009). A Novel Approach to Generate Test Cases from UML Activity Diagrams. *Journal of Object Technology*.

Lano, K. (2009). *UML 2 Semantics and Applications*. John Wiley & Sons.

Maiden, N. A., Manning, S., Jones, S., and Greenwood, J. (2005). Generating requirements from systems models using patterns: a case study. *Requirements Engineering*.

Nicolás, J. and Toval, A. (2009). On the generation of requirements specifications from software engineering models: A systematic literature review. *Information and Software Technology*.

Object Management Group (OMG) (2015). OMG Unified Modeling Language (OMG UML), Version 2.5.

Sikora, E., Tenbergen, B., and Pohl, K. (2012). Industry needs and research directions in requirements engineering for embedded systems. *Requirements Engineering*.

Störrle, H. (2004). Semantics of UML 2.0 Activities. In *Symposium on Visual Languages and Human-Centric Computing*.

Usman, M. and Nadeem, A. (2009). Automatic Generation of Java Code from UML Diagrams using UJECTOR. *Journal of Software Engineering and Its Applications*.

Vogelsang, A., Eder, S., Hackenberg, G., Junker, M., and Teufl, S. (2014). Supporting concurrent development of requirements and architecture: A model-based approach. In *Conference on Model-Driven Engineering and Software Development*.

Weber, M. and Weisbrod, J. (2002). Requirements Engineering in Automotive Development - Experiences and Challenges. In *International Conference on Requirements Engineering*.