Towards Automated fUML Model Verification with Petri Nets

Francesco Bedini, Ralph Maschotta, Alexander Wichmann and Armin Zimmermann
Systems and Software Engineering Group, Technische Universität Ilmenau, Ilmenau, Germany

Keywords: fUML, Activity Diagram, Petri net, Transformation, Model-to-Model, QVTo.

Abstract: One of the goals of the Foundational UML Subset (fUML) is a consistent and well-defined execution of UML activity diagrams. However, the specification is not done in a formal mathematical model and leaves room for implementation-specific tool details. This paper shows how this may lead to problems for concurrent program semantics. To this end, the paper introduces a transformation and basic analysis methods for activity diagrams under the current fUML sequential execution semantics. The analysis is conducted using Petri nets, which are mathematical models with a graphical representation to describe distributed systems. There are numerous well-established analysis methods to validate specific desirable properties of a concurrent program including liveliness, the absence of deadlocks, fairness, mutual exclusion, and detection of unreachable states. In this paper, we show that the intuitive translation to Petri nets does not fit the current fUML execution implementation; therefore, we introduce a new model-to-model transformation realized with QVTo, that translates a set of the most used fUML elements to Petri nets. Moreover, we propose methods to analyze the models with the tool TimeNET.

1 INTRODUCTION

UML activity diagrams are one of the UML's behavior diagrams; they describe how states and steps of a process can be viewed as a flow of control and data exchange between their different actions. The specification of activity diagrams changed considerably over time. Initially, until version 1.5 (from now on: UML 1.x), the semantics was inherited from state machines (Arlow and Neustadt, 2002). UML 2.0 (from now on: UML 2.x) introduced a different token-driven semantics inspired by Petri nets and ultimately abandoned the concept of states inside activity diagrams (Störrle and Hausmann, 2005).

With this fundamental change, it has become possible to model more advanced parallel executions (and multiple calls to the same activity) that were not possible before in UML 1.x.

UML 2.x brought new elements such as object pins, merge nodes and flow final nodes to the modelers' palette. However, the old syntax remained unchanged for the most part, but yet with a completely different execution semantics. Unfortunately, these differences led to models that were syntactically valid for both UML versions but having completely different results under certain circumstances (Störrle, 2005).

Behavioral semantics and execution analysis of dynamic UML diagrams has attracted a lot of research interest over the years. Partially unclear semantics were found and tried to overcome. Possible approaches may either analyze properties of a model-specified program directly or derive results via an intermediate model and corresponding transformations (Eshuis and Wieringa, 2004).

The introduction of a *Foundational UML Subset* (fUML) in 2011 aimed at a predictable interpretation of the execution semantics of a subset of UML activity diagram elements; a token-based execution flow was introduced to precisely describe how the single elements behave. One goal was to ensure that different execution tools will produce the same consistent output from running the same model. The fUML specification assures the execution predictability due to a sequentialized execution of the models (OMG, 2017a). However, the fUML specification foresees but does not strictly specifies how a parallel architecture should execute activity diagrams. This lack of preciseness may lead to problems that only become evident when executing concurrent flows sequentially, or when different tools will support concurrent execution differently, as will be presented in Section 2.3.

The fUML execution engine (OMG, 2017a) is implemented to always iterate through all the output edges of each element, and sending a token in a well-defined fixed order. If the next element on the *target*

Bedini F., Maschotta R., Wichmann A. and Zimmermann A.
Towards Automated fUML Model Verification with Petri Nets.
DOI: 10.5220/0007371403000308
In *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 7th International Conference on Model-Driven Engineering and Software Development)*, pages 298-306
ISBN: 978-989-758-358-2

side of the edge is ready to execute (i.e., it already received all inputs control and data tokens), it will do so; otherwise, another token will be sent to the next output edges, if any. The order of the output edges is not fixed and is difficult to predict, as it may depend on how the user created the elements in the model, how the modeling tool saved them, and also on the specific execution engine implementation. Letting the parallel execution semantics of a model depend on such technical details, that the modeler is often unaware of, will lead to misconceptions and unexpected program behavior and thus counteract the intention of fUML.

As most of personal or laptop computers contain at least two CPU cores nowadays, it is desirable and foreseeable that the fUML execution engine will execute independent activity diagrams actions in a parallel way in the future. This will help to reduce execution times, which are still significantly higher for such models in comparison to conventional programs (Bedini et al., 2017).

The fUML specification just briefly mentions the parallel execution of diagrams, but foresees it and its possible issues by, for example, including a *mustIsolate* property for denying interleaved execution by stating that *if an execution tool uses locking to implement isolation, then it also must provide some means to detect implementation-specific deadlock conditions and recover from them* (OMG, 2017a).

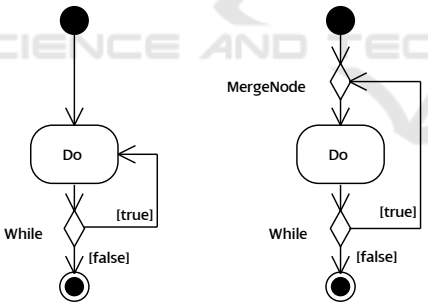


Figure 1: Two realizations of a *do-while* loop as an activity diagram. Left: UML 1.x, Right: UML 2.x.

Figure 1 shows how a *do-while* loop would have been realized in the old and new syntax. If the UML 1.x version would be run by an fUML execution engine, the execution would hang with a deadlock after sending the first token to the *Do* action, because it would wait for a second token coming from the decision node before being able to execute.

Moreover, the fUML specification, being provided as a Java-like implementation, does not provide a good mathematical basis for analysis techniques (Laurent et al., 2014).

For these reasons, a verification tool for fUML models and the execution engine is needed. With fUML and UML's activity diagrams based on Petri net's token paradigm, and aiming to take advantage of the already existing literature and methods about Petri nets verification (Murata, 1989), we decided to bring the fUML diagrams back to the origin by transforming them to Petri nets. In this way, we can perform various analyses to check that specific properties hold within the structure of the model and during its execution, using the Petri net's mathematical formalism as a reliable formal basis.

One result of this paper is a Model-to-model (M2M) transformation that respects the fUML execution semantics and provides the foundations for a lint-like validation tool based on Petri net analysis for fUML models. This tool will make the modeler aware of design pitfalls and semantic ambiguities by detecting possible problems that may arise when executing the diagram sequentially or in parallel.

The following section shows the methodology, sets the analysis' goals and briefly surveys different Petri net analysis techniques. Some related work is covered along the way. Section 3 shows the transformation rules specific to fUML and how the transformation has been realized. Section 4 gives an application example, while results and future work are discussed in the conclusion section. Finally, the Appendix covers the proposed set of single transformation rules.

2 METHODOLOGY

This section at first introduces the issues that we aim to detect in activity diagrams, followed by an overview of the primary analysis techniques in Petri nets that may be used for computing them. We introduce a model-to-model transformation afterwards and cover some assumptions about the original activity diagram to allow such a transformation.

2.1 Analysis Goals

Similar to the concept of *soundness* in the area business process modeling (van der Aalst et al., 2000), we want to make sure that the modeled activity diagrams have a safe and predictable flow.

For achieving this goal, we would like to analyze activity diagrams to prove or confute some of the following conditions:

Deadlock is said to happen if the model reaches a configuration (state) in which no action can *fire* (i.e., execute).

Liveness means that the system will keep running, requiring at least the absence of deadlocks, or the

absence of starvation in some stricter definitions. Both full and partial liveness covering the whole or only parts of a model may be of interest.

Unreachable Regions — a set of elements that will not get executed under certain circumstances, for example, because they are located after a never-ending loop.

Problems with concurrent programs falling into these areas are often rooted in synchronization problems and caused by erroneous modeling. The advantage of looking for issues on a Petri net rather than on the activity diagram directly is that the mentioned properties are well researched for Petri net models, and efficient algorithms for their computation exist (Sifakis, 1978; Girault and Valk, 2003). Moreover, Petri nets have been introduced for the description of concurrent systems with synchronization, and should thus be capable of capturing control flow information of activity diagrams in a natural way.

2.2 Petri Net Analysis

This section will describe possible ways of mapping the issues listed in Section 2.1 to Petri net properties and detecting them with algorithm types known for Petri nets.

The mentioned properties of an activity diagram can be mapped to similar Petri net properties, provided that the behavior of the transformed model is identical:

Deadlocks a deadlock occurs in a Petri net when no transition can fire due to the lack of (one or more) input tokens.

Liveliness Petri net transitions and nets have several levels of liveliness (L_0: dead - L_4: live) (Girault and Valk, 2003) that can be mathematically proved.

Unreachable Regions a set of transitions that will not fire because of missing tokens in a state reachable by the initial net marking.

Moreover, it is possible to analyze the net's **boundedness** to check if the amount of tokens in the net remains bounded, which may otherwise point at infinite loops or wrong recursion steps.

Provided that a behavioral similar model has been created, the next step is to compute the properties with existing analysis methods. Reachability analysis, which consists of enumerating and analyzing all reachable states of a net, does not scale for bigger models or unbounded networks with an unlimited number of states. This issue is known as the *state space explosion* problem (Valmari, 1998).

A linear algebraic analysis approach would lead to a similar problem, resulting in many linear equations to consider.

It would be then possible to apply some transformation techniques to simplify the Petri net making the analysis process more straightforward, but then the difficulty would be finding the reducible subnetworks and then backtracking the error to the original element.

Synthesis methods for reducing the size of Petri nets are NP-complete (Badouel et al., 2015) and cannot cope with strongly coupled subsystems.

Simulating the Petri net often requires long run times, and does not assure that all possible markings are covered. This well-known principal problem of simulation algorithms is similar to the test coverage problem.

Graph theoretical analysis (for example, looking for siphons and traps (Murata, 1989)) are inexpensive but may detect false-positives too. For instance, activity initial and final nodes would be detected as siphons and traps, respectively.

For these reasons the authors argue that a graph theoretical analysis should be preferred over the other methods, as it is less expensive to compute and provides satisfying hints about possible problems in the net. Those hints can then be analyzed individually with a heuristic to filter out possible false positives.

2.3 Activity Diagram Transformation to Petri Nets

With the UML activity diagram specification based on Petri nets semantics (OMG, 2017b), the transformation of activity diagrams elements to Petri net snippets should be quite natural from a conceptual point of view. Similar transformation approaches have been reported in the literature, although with different goals.

The transformation can be performed to evaluate program performance as in (Distefano et al., 2011) and (López-Grao et al., 2004), where UML 1.x activity diagrams are transformed to stochastic Petri nets to evaluate their performance. This technique can also be applied to compute other properties of the modeled system. For example, system execution time and number of resources that the system would consume are simulated in (Andrade et al., 2009). This approach is based on SysML activity diagrams transformed into Time Petri nets to validate the requirements of embedded systems.

Our goal and approach are more similar to the one introduced in (Agarwal, 2013), which proposed a method for transforming the essential elements of

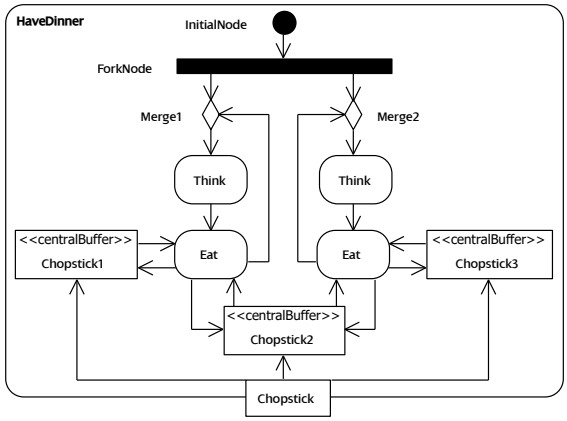


Figure 2: An activity diagram modeling the dining philosopher problem with only two philosophers.

UML 2.x activity diagrams. The goal there is to analyze the diagrams' liveliness, boundedness, and reachability for verification purposes. The transformation methods we used are also similar to the *intuitive* ones covered in (Staines, 2008) and (Störrle, 2004). The transformation rules we used are displayed in the appendix for completeness.

2.3.1 fUML Specification Issues

To demonstrate the issues which arise by applying the existing transformations to fUML activity diagrams, let us consider the well-known *Dining Philosopher* problem (Andrews, 1991) as a kind of benchmark for concurrent system design. Figure 2 shows, for simplicity, a section with just two philosophers (as models for concurrent processes) who are having dinner.

For simplicity, we allow in the model the philosophers to take both chopsticks with one atomic action, such that at least deadlocks cannot happen according to the requirements of deadlocks in concurrent systems. The execution starts and two identical flows start to run. Each philosopher is thinking at the beginning. Then, if both chopsticks are available, they can eat (one at a time) and go back to thinking afterwards.

The changes introduced by the fUML specification regarding the fixed order of execution of the model profoundly changed the semantics depicted initially by UML 2.x, in a way that can be misleading for some modelers.

Due to the *depth-first* approach and the strict ordering intrinsically modeled in the fUML specification (inside the function `ActivityNodeActivation::sendOffers`), the first merge node that gets executed keeps running forever, leading the other philosopher to starvation. Which one is the first depends on the order how the fork node edges have been saved or on the order how the execution engine internally stores them, which is a low-level technical issue that should not influence the model behavior.

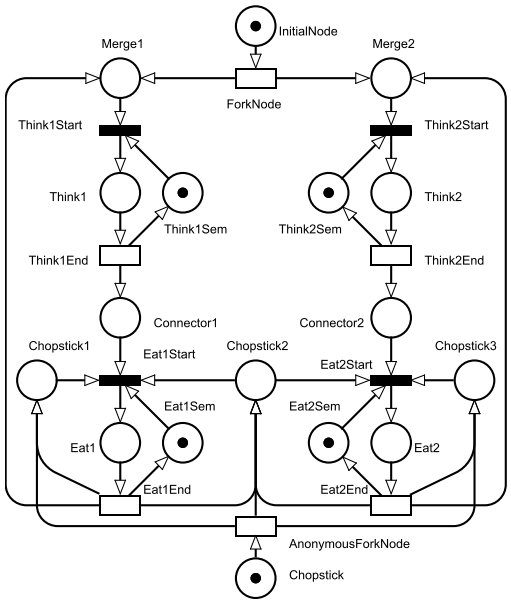


Figure 3: A Petri net representing the activity diagram in Figure 2, obtained using the intuitive transformation.

With just two philosophers, it would be possible to solve this problem by using a decision node that randomly picks one of its two outgoing edges instead of the fork node. Unfortunately, this solution would not scale with more philosophers (only one would be allowed to eat at any time), and it requires the modeler to include a scheduling detail in its model instead of focusing on the high-level concurrent design of the problem.

UML 2.x specified that *concurrent execution means that there is no required order in which the nodes must be executed; a conforming execution of the Activity may execute the nodes sequentially in either order or may execute them in parallel* (OMG, 2017b). fUML adheres to this specification by executing one of them first but never gets to execute the second one until the first one finishes (and that is never going to happen in our example).

If we would transform the activity diagram in Figure 2 to a Petri net following the *intuitive* transformation based on the UML 2.x specification, we would obtain the diagram shown in Figure 3, which would be correct concerning the original semantics of UML 2.x, but not according to the new *sequentialized* execution semantics of fUML.

Analyzing this Petri Net would not lead to discovering any problem. The model executes as expected, both processes execute concurrently, and both

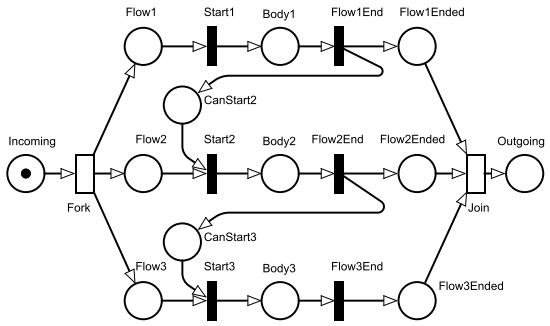


Figure 4: A Petri net realizing fUML a *fork and join* construct with a sequential execution. The three *body* places are just placeholders; in reality, actions are not transformed as a single place, but as a set of elements (as presented in the Appendix).

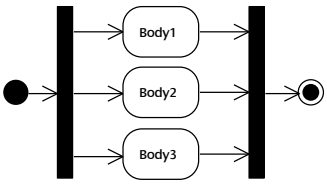


Figure 5: An activity diagram showing a fork and join of three flows.

philosophers will eventually be eating. However, it does not resemble the specified fUML behavior and thus does not show problems arising from it. For this reason, we introduce a different kind of transformation, to resemble fUML's strict sequential execution of fork node flows.

2.3.2 Sequential Transformation of Fork Nodes

This section proposes an fUML-conform transformation technique for transforming fork nodes to Petri nets. The proposed transformation replicates the fUML sequential execution of fork nodes outgoing flows, to be able to detect problems that would happen during the sequential execution of activity diagrams.

Figure 4 shows our proposed transformation for the activity diagrams in Figure 5.

We introduce two additional immediate transitions for each flow that should fire at the beginning and the end of the flow. For this to happen, we need one additional place (*CanStart#*) for all flows (except the first one), that shall receive one token once the previous flow finishes its execution. The *FlowEnd* transition takes care of placing this token.

In case a flow does not end (for example when it contains an infinite loop), the *CanStart* place of the following flow will not receive a token, clearly marking the remaining flows as unreachable inside the Petri net. Moreover, looking for the end (or ends) of the flows allows us to locate critical sections or loops in the model already during the transformation phase.

It is evident that the *Flow2* and *Flow3* places and the *Fork* transition itself are redundant and could be left out in this diagram without missing anything in the final analysis. However, to make the recognition of the different flows simpler and their structure more systematic, we have decided to still include them in the transformation. Additionally, when those places are identified as traps, we can warn the user more precisely about which flow will encounter problems.

Finding the final element of a flow is not trivial and will be solved in future work. At this time, we can simplify this task by saying that a *join* or *flow final* or *activity final* nodes would resemble, for example, an obvious end of the flow.

Another problem that arises is that the order used during the execution is implementation-specific and depends on different factors, such as how the elements get saved inside the fUML model, and how the execution engine reads and stores the edges (in a FIFO or a LIFO data structure, for example).

For example, in Figure 6 a peculiar but syntactically valid fork node with three actions is shown. Depending on which flow gets executed first, it is possible to obtain several different results:

A executes first: the action A executes once, and the activity ends, reaching the activity final node.

B executes first: the action B executes once and then, depending on the internal ordering, either A gets executed once, or C gets executed in an infinite loop with B (B, C, B, C. . .).

C executes first: results in an infinite loop of C actions (C, C, . . .).

These examples show that although the fUML goal was to be specific enough to provide a precise semantic for consistent execution flows, implementation-specific details can still produce different results while running the same model. To use the proposed sequentialized transformation, the M2M transformation either requires some extra knowledge

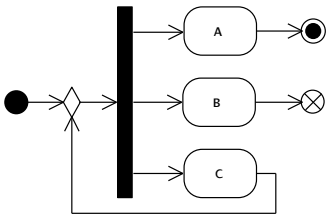


Figure 6: An activity diagram showing one fork node with three outgoing flows. Its execution heavily depends on the specific execution engine implementation.

about the implementation-specific peculiarities of the fUML execution engine or should produce all possible combinations and analyze all of them.

2.4 Transformation Assumptions

For the analysis, we require the following assumptions or limitations for activity diagrams to allow them to be transformable:

Assumption 1. All actions must terminate and produce the outputs they are expected to provide on their output edges, even if empty.

This restriction is necessary because the transformation loses some information about the behavioral semantics of the original model, which is possible to model in different ways (for example as programming code inside a function behavior), and it is not possible to easily transfer it into a Petri net. For this reason, our analysis must assume that actions never hang and always produce a token on all their outgoing edges. Informally, the transformation cannot take a look "inside" the actions, and thus the significant behavior must be clear from the model structure.

Assumption 2. Actions should not modify attributes that may get used by other actions which may run concurrently during their execution.

For the same reasons of Assumption 1, all actions must resemble an atomic set of operations. Activity diagrams are high-level descriptions of a process, and only elements that are explicitly present in a model can be analyzed. Therefore it is important to not hide too much logic behind actions, which can be expressed using other activity diagrams elements.

Assumption 3. Activity diagrams may consist of a set of supported elements only. This set consists of activity initial and final nodes, activity input and output parameters, flow final nodes, decision nodes, merge nodes, fork and join nodes, central buffer nodes, datastore nodes, actions with input and output pins and loop nodes.

If the diagrams contain unsupported parts such as event-related elements (i.e., send/receive signal actions and time events), they will not be transformed, and the analysis would take place on an incomplete diagram probably leading to wrong or at least only partial results.

Assumption 4. The input model is syntactically valid, which implies that all elements are correctly connected with control or object flows. As the transformation considers both data and control flows as one single kind of tokens, it is essential that the original model allows to transfer them correctly.

3 REALIZATION

To implement the transformation, we have used *operational QVT* (QVTo), whereas for performing the Petri net analysis, we are using the existing tool TimeNET (Zimmermann, 2017). QVTo is an OMG standard using an OCL-like imperative language, being still actively developed and well integrated into Eclipse (Tikhonova and Willemse, 2015). It works following the Query/View/Transformation paradigm, requiring to first *selecting* (query) from the original model the elements which will be transformed, then define how they should *look like* (view) in the output model by defining a *transformation* that realizes it.

The exogenous model-to-model transformation from activity diagrams to Petri nets requires the source and the destination meta-models. For UML we used the UML ecore meta-model available from the core Eclipse Modeling Framework (EMF), whereas for the Petri net meta-model we have generated the ecore meta-model from TimeNET's internal XML schema files (XSD) specifying stochastic Petri nets.

In TimeNET, this schema does not only describe *Extended Deterministic and Stochastic Petri nets*, but represents a model class containing the modeling power of several well-known sub-classes like GSPNs, DSPNs, and eDSPNs (Zimmermann, 2007) allowing different transition firing delay types. This extended set allows us not to be restricted *a priori* to a single class of Petri nets, leaving us the possibility to extend or switch to a different subclass of Petri net when new analysis methods are needed.

Once the input and output meta-models are available, the input model can be provided to a transformation tool, which will execute an ordered set of transformation rules (also known as mappings) to produce a Petri net model.

QVTo uses a hierarchical waterfall approach, where a linear sequence of steps obtains the transformed model. The transformation operates as illustrated in Figure 7. However, due to QVTo's limitations, the transformation is implemented sequentially and operates as follows:

1. Transform firstly all the single activity diagram's nodes (for the transformation details, refer to the Appendix).
2. Transform all edges between elements:

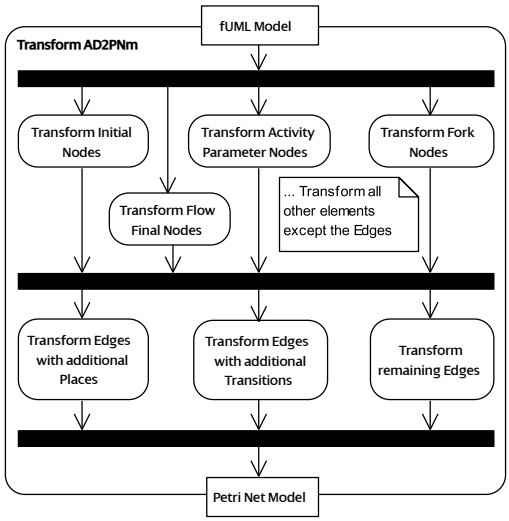


Figure 7: Activity diagram showing the required steps for transforming an fUML activity diagram to a Petri net with QVTo.

(a) transform all edges which require an extra transition (between two places);
(b) transform all edges which require an extra place (between two transitions);
(c) transform the remaining edges (between a place and a transition, and vice versa).

With the transformation process in place, the activity diagrams can be now translated to Petri nets. This transformation is not complete and loses a part of the original model semantics, for instance the actions' function behaviors which are internally defined as source code. On the other hand, the transformation allows us to perform more advanced analysis on the general structure of the diagram.

4 EXAMPLE

This section describes how the proposed sequential Petri net transformation can detect starvation in the philosophers problem previously described in Section 2.3.

It is possible to apply different techniques to analyze the resulting Petri net shown in Figure 8. For example, a structural analysis of the net shows that the sets of places {Merge1, Think1, Connector1, Eat1} and {Merge2, Think2, Connector2, Eat2} are traps, denoting the presence of possible never-ending loops. Petri net traps are sets of places which may receive additional tokens, but will never allow them to leave the trap.

Using the TimeNET tool, the reachability graph (Figure 9) can be computed. It shows that there is an

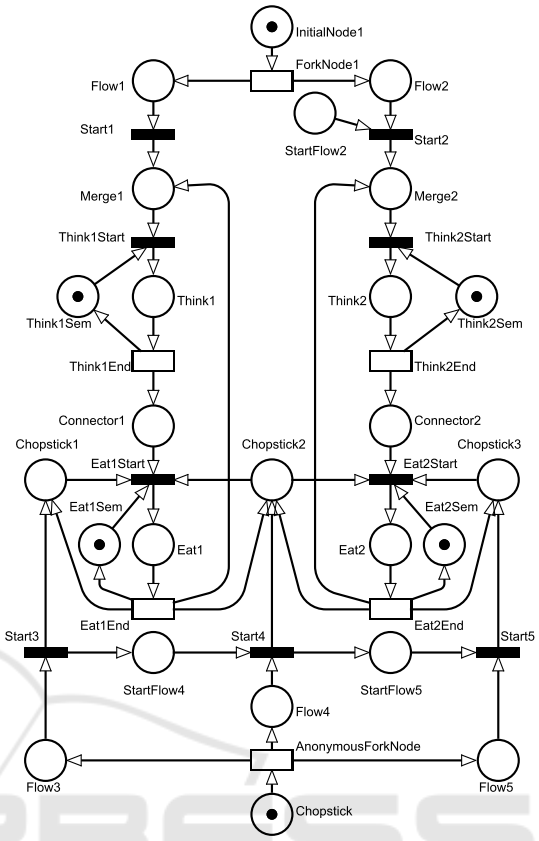


Figure 8: A Petri net showing the result of the proposed sequentialized transformation of the fUML model in Figure 2

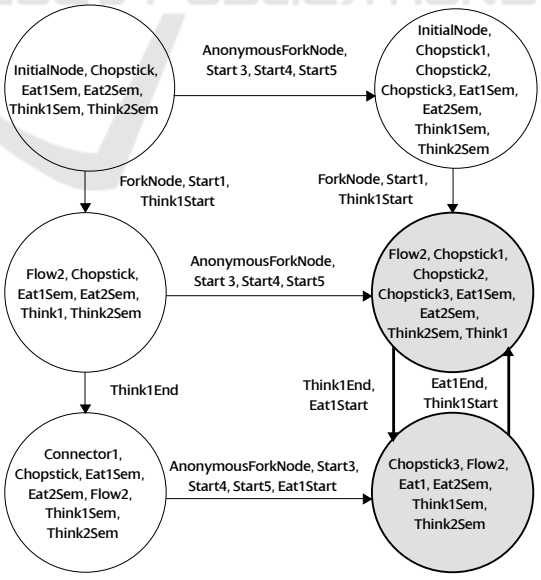


Figure 9: The reachability graph of the Petri net shown in Figure 8. Inside the circles, places who hold exactly one token are listed, while transitions causing state changes are shown on the arcs. The grey elements visualize a loop.

infinite loop between two states, which hints to the same (potential undesired) condition in the original diagram. Moreover, as some transitions are never firing in the reachability graph, some parts of the model containing the missing transitions are unreachable from the initial marking (*Think2Start* and *Eat2Start* in the example).

The shown steps demonstrate that difficulties in transforming an activity diagram into a Petri net arise from the incomplete specification of such diagrams, but that execution-specific issues can be detected with a correctly extended Petri net that incorporates the sequential fUML execution.

5 CONCLUSION

In this paper, we have developed the foundation for a lint-like fUML activity diagram verification, which can be used to detect and warn modelers about errors or potentially dangerous anomalies in their diagrams describing concurrent programs. This approach is based on two types of M2M transformations to Petri nets: 1) an intuitive one similar to the existing literature to tackle conceptual problems, and 2) an fUML-conform one introduced here to find problems caused by the sequential execution semantics specified by the fUML. In particular, our method with its different transformations, aims to detect synchronization problems that would occur when the diagrams are executed both concurrently and sequentially.

The sequential transformation of fork nodes allowed us to verify not only models that were obviously wrong (for example with disconnected elements), but also models without such problems that in reality lead to a peculiar execution which is hard to predict without expert knowledge of the execution engine implementation.

As a side effect, the paper points out current limitations and potentially dangerous situations of the current fUML specification, that would only become evident when the execution happens concurrently or with a different static execution order.

5.1 Future Work

The next steps consist of finding a suitable algorithm for correctly transforming all possible constellations of fork node flows; i.e., finding or deciding the absence of the end of each flow. Once a reliable method is defined, it will be possible to implement the sequential transformation of fork node flows.

Further work will include parallelization of the fUML C++ execution engine developed in our group (MDE4CPP, (Bedini et al., 2017)), automation of the M2M transformation, and the execution of the analysis techniques which are relevant to the input model.

The transformation implementation is available online[1] and distributed under the MIT license.

REFERENCES

Agarwal, B. (2013). Transformation of UML Activity Diagrams into Petri Nets for Verification Purposes. *International Journal Of Engineering And Computer Science*, 2(3):798–805.

Andrade, E., Maciel, P., Callou, G., and Nogueira, B. (2009). A methodology for mapping sysml activity diagram to time petri net for requirement validation of embedded real-time systems with energy constraints. In *2009 Third International Conference on Digital Society (ICDS)*, pages 266–271.

Andrews, G. R. (1991). *Concurrent programming: principles and practice*. Benjamin/Cummings Publishing Company San Francisco.

Arlow, J. and Neustadt, I. (2002). *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*. Pearson Education, Inc., 2 edition.

Badouel, E., Bernardinello, L., and Darondeau, P. (2015). *Petri net synthesis*. Springer.

Bedini, F., Maschotta, R., Wichmann, A., Jäger, S., and Zimmermann, A. (2017). A model-driven fuml execution engine for C++. In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2017, Porto, Portugal, February 19-21, 2017.*, pages 443–450. SciTePress.

Distefano, S., Scarpa, M., and Puliafito, A. (2011). From UML to Petri Nets: The PCM-Based Methodology. *IEEE Transactions on Software Engineering*, 37(1):65–79.

Eshuis, R. and Wieringa, R. (2004). Tool support for verifying uml activity diagrams. *IEEE Transactions on Software Engineering*, 30(7):437–447.

Girault, C. and Valk, R. (2003). *Petri Nets for System Engineering*. Springer.

Laurent, Y., Bendraou, R., Baarir, S., and Gervais, M.-P. (2014). Formalization of fuml: An application to process verification. In *International Conference on Advanced Information Systems Engineering*, pages 347–363. Springer.

López-Grao, J. P., Merseguer, J., and Campos, J. (2004). From UML Activity Diagrams to Stochastic Petri nets: application to software performance engineering. *ACM SIGSOFT Software Engineering Notes*, 29(1):25–36.

Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580.

OMG (2017a). fUML 1.3 Specifications. https://www.omg.org/spec/FUML/1.3/PDF. online.

[1] https://github.com/MDE4CPP/AD2PN

OMG (2017b). Unified Modeling Language, Specification version 2.5.1. https://www.omg.org/spec/UML/2.5.1/PDF. online.

Sifakis, J. (1978). Structural properties of Petri nets. In Winkowski, J., editor, *Mathematical Foundations of Computer Science*, pages 474–483. Springer Verlag.

Staines, T. S. (2008). Intuitive Mapping of UML 2 Activity Diagrams into Fundamental Modeling Concept Petri Net Diagrams and Colored Petri Nets. In *2008 15th Annual IEEE International Conference on Engineering of Computer Based Systems (ECBS)*, pages 191–200.

Störrle, H. (2004). Semantics of control-flow in uml 2.0 activities. In *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 235–242.

Störrle, H. (2005). Semantics and verification of data flow in uml 2.0 activities. *Electronic Notes in Theoretical Computer Science*, 127(4):35 – 52. Proceedings of the Workshop on Visual Languages and Formal Methods (VLFM 2004).

Störrle, H. and Hausmann, J. H. (2005). Towards a formal semantics of uml 2.0 activities. In *Software Engineering*.

Tikhonova, U. and Willemse, T. (2015). Designing and describing qvto model transformations. In *2015 10th International Joint Conference on Software Technologies (ICSOFT)*, volume 1, pages 1–6.

Valmari, A. (1998). *The state explosion problem*, pages 429–528. Springer Berlin Heidelberg, Berlin, Heidelberg.

van der Aalst, W., Desel, J., and Oberweis, A. (2000). *Business Process Management — Models, Techniques, and Empirical Studies*. LNCS 1806. Springer Berlin Heidelberg.

Zimmermann, A. (2007). *Stochastic Discrete Event Systems — Modeling, Evaluation, Applications*. Springer, Berlin Heidelberg New York.

Zimmermann, A. (2017). Modelling and performance evaluation with TimeNET 4.4. In Bertrand, N. and Bortolussi, L., editors, *Quantitative Evaluation of Systems - 14th Int. Conf., QEST 2017*, pages 300–303, Berlin, Germany.

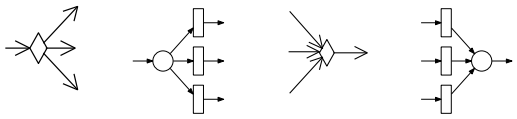
APPENDIX

fUML Activity Diagram to Petri Net Transformation

This appendix shows how the QVTo transformation translates the single fUML elements to Petri nets using the intuitive transformation.

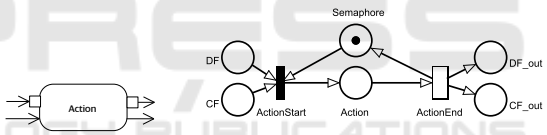
Activity Initial Node / Activity input parameter.

Flow Final node / Activity Output Parameter. An Activity Final node may be translated in the same way, but it would not have the same semantic meaning as in an activity diagram.



Decision/Merge nodes with 3 flows.

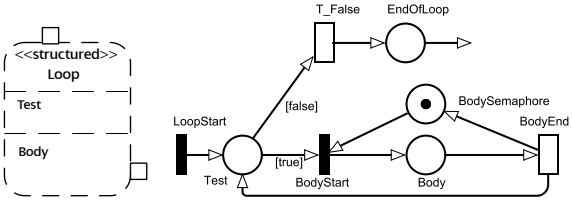
Fork/Join nodes with 3 flows (intuitive transformation).



A Call Behavior/Operation Action, with incoming and outgoing data and control flows. The semaphore allows to execute the action multiple times in a loop with the correct number of input parameters.

A Central buffer node.

A Data Store Node with one input and one output flow.



A Loop Node. According to the fUML specification, its `Setup` section must be empty.