Operational Semantics of UML Activity Diagram: An Application in Project Management

Eugene Syriani
esyriani@cs.ua.edu

Department of Computer Science
University of Alabama
Tuscaloosa AL, U.S.A.

Hüseyin Ergin
hergin@crimson.ua.edu

***Abstract*—With its recent adoption by the International Organization for Standardization, we foresee that UML will be systematically used for object-oriented modeling in industry. UML activity diagrams have been typically used to model software and business processes. Due to its semi-formal semantics and high complexity, its advanced constructs such as expansion regions, interruptible regions, object nodes, time events, and compound activities are rarely used in practice. There has been significant work on formalizing UML activity diagrams in terms of its semantic domain: Petri net. However, none address the recent advanced constructs it offers. In this paper, we define the semantics of UML activity diagram using a rule-based model transformation. Verification and validation of the UML activity diagram model is then achieved by simulating and analyzing the Petri net model. We illustrate our technique by using an extension of UML activity diagram to facilitate project management tasks such as scheduling, cost estimation, and resource allocation.**

***Keywords*-UML activity diagram; Petri net; model transformation.**

I. Introduction

UML activity diagram (AD) is a very widespread notation in requirement engineering, especially used to model software and business processes [1], [2]. In 2001, UML 1.4 informally defined AD with a state machine-like semantics. Since the UML 2.0 in 2005, AD was redefined with semantics in terms of Petri nets (PN) to essentially allow parallel flows. Recently in April 2012, the International Organization for Standardization (ISO) adopted UML 2.4.1 which included minor changes to AD [3]. Our work is based on that reference. Although the semantics of AD is described in natural language, a precise formal semantics is still missing. This hampers the understanding, analysis, and interpretation of AD instances. This is of paramount importance in requirement engineering where users are often business analysts and not software engineers.

Many works in the literature (*e.g.,* [4], [5], [6], [7], [8], [9]) have formalized the semantics of UML2 AD, more specifically in terms of PN and variations of it. However, they are all restricted to mostly basic AD elements (action, fork, join, branch, merge, decision) and do not take into considerations more complex constructs such as expansion regions, interruptible regions, object nodes, time events, and compound activities, which is the main reason why they are rarely used in practice. Furthermore, with its recent ISO adoption, we foresee that AD will be systematically used for requirement engineering in industry. We therefore propose a pragmatic definition of the operational semantics of the complete AD formalism in terms of PN. We use a rule-based model transformation approach to define the translation process. Consequently, we execute AD models by simulating the underlying PN model.

In Section II, we provide the relevant meta-models of AD and PN for semantic translation and describe the transformation. In Section III, we propose an extension of AD as a domain-specific modeling language for project managers and show how the transformation can be used to facilitate his tasks. Section IV discusses related work and we conclude in Section V.

II. The UML 2 Activity Diagrams Formalism

In this section, we describe a precise definition of the semantics of AD in terms of PN.

A. Language Definition

The input of the translation process is the AD language whose meta-model is depicted in Figure 1 and the output is the PN language whose meta-model is depicted in Figure 2. The translation focuses on the essential AD elements with semantics. That is, we have purposely omitted those elements in the ISO reference only used for syntax. We also do not take into consideration the interaction between AD elements and the behavior of the objects flowing (from the BasicBehavior package) and treat objects as simple tokens. Additionally, we have encapsulated all ExecutableNodes under StructuredActivities since they define specific interactions with objects. The AD meta-model in Figure 1 is therefore an abstraction of the full ISO version, focusing on the elements with *essential operational semantics* (or *token rules* in [3]). Note that the meta-model is augmented with additional constraints such as: final nodes and accept signal nodes only have incoming activity edges, whereas initial nodes and signal nodes only have outgoing edges.

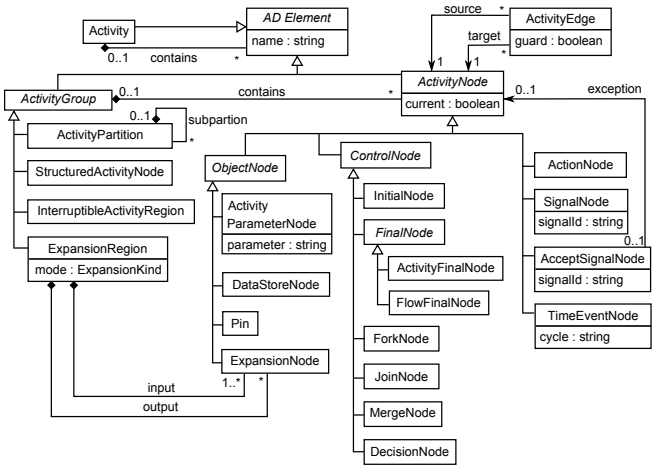

Figure 1. The meta-model of UML activity diagrams.

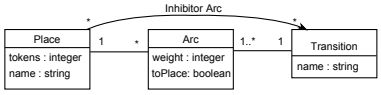

Figure 2. The meta-model of Petri nets.

Also, activities can only be connected via object nodes. Additionally, if multiple edges are outgoing from an action, then object nodes must be used to distinguish among them. Finally, the guard condition in the activity edge is used only for selecting the suitable branch at decision nodes.

The reference manual claims that the semantic domain of AD is PN. Therefore we tried as much as possible to satisfy that claim by using pure place/transition nets. However, as the subsequent rules will show, inhibitor arcs are required at minimum to handle complex AD elements such as expansion and interruptible regions. As defined in [10]: an inhibitor arc connects a place to a transition and is represented by a dashed line terminating with a small circle. It disables the transition when the input place has at least one token and enables it otherwise if other input places have at least one token per arc weight. This makes PN Turing complete and, by transitivity, AD is Turing complete. Figure 2 depicts the meta-model of PN we considered.

B. Semantic Translation

The semantic translation is modeled as a model transformation mapping AD elements to behaviorally equivalent PN counterparts. The modeling language and test models are defined using AToM^3 [11]. The transformation is implemented in Py-T-Core [12], a domain-specific graph transformation language. It consists of a set of rules and a control structure to schedule their execution. We chose to define the semantics of AD using rule-based descriptions as they allow specifying the transformation as a set of operational rewriting rules instead of using imperative programming languages.

The rules are shown in Figures 3–8. We use the visual concrete syntax of MoTif [13] where the central compartment is the left-hand side (LHS), the compartment on the right of the arrow head is the right-hand side (RHS) and the compartment(s) on the left of dashed lines are the negative application condition(s) (NAC). The LHS defines the pre-condition pattern that must be found in the input model to apply the rule. The NAC defines a pre-condition pattern that shall not be present, inhibiting the application of the rule. The right-hand side (RHS) imposes the post-condition pattern to be found after the rule was applied. Numeric labels are used to uniquely identify different elements across compartments. Generic links are used as explicit traceability links, connecting any two elements from any meta-model.

This transformation is organized in *phases*. Each phase consists of a set of rules that are executed in an arbitrary order until they are all applied. The 11 required phases are shown in Table I. A rule annotated with a ^{F} is applied once for all matches of the pre-condition pattern found in the model. A rule annotated with a * is applied iteratively as long as matches are found. We now describe each rule following their execution order.

1) Basic Elements: Figure 3 depicts the rules of the first phase, translating basic elements to their PN counterparts. Each action node is mapped to an *entry* transition, a *processing* place and an *exit* transition, as in [14]. The entry transition initiates the execution of the action and the exit transition terminates its execution. This representation satisfies the "eventual execution of an action upon receiving input tokens". Fork, join, object, and activity parameter nodes are mapped to a single transition since they are *transient* in AD [8]. Activity final, flow final, and signal nodes are only mapped to an entry transition and processing place, since they mark the end of an activity. Conversely, time event and accept signal nodes are mapped to a processing place and an exit transition, since they cannot be the target of edges. The initial node is mapped in the same way, but with a token added in the processing place to initiate the execution. Decision nodes are simply mapped to a set of transitions. Each branch is mapped to a different transition to force that only one of them becomes fireable at run-time. Merge nodes are treated similarly, but instead of creating branches, they merge them. Note how in the DecisionMapping rule, the activity node element will be matched to an element from any of its sub-types.

2) Edge Connections: Figure 4 depicts the rules of the following two phases that handle edge connections. In general, each activity edge (and exception edge) is mapped to a place between the transitions of its source and target AD elements. It may very well happen that some AD elements are connected to more than one element. In this case, the rules of the third phase cover the remaining edges. Since the ActivityEdgeBetweenObjectNode&ActionMapping rule is applied iteratively, the case where actions have multiple

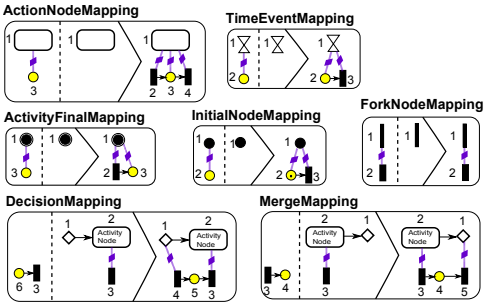

Figure 3. The first set of rules: Node mapping.

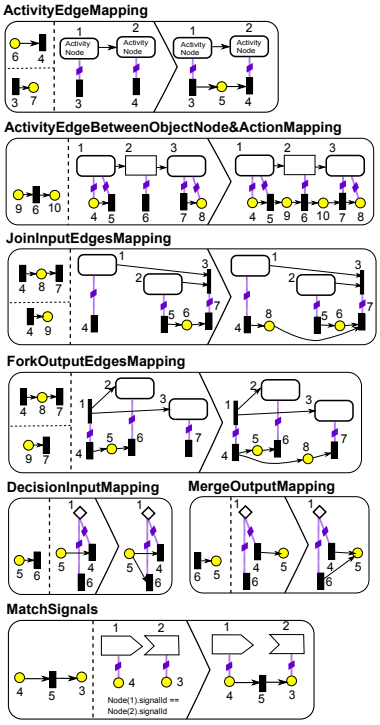

Figure 4. The second set of rules: Edge mapping.

object nodes as input and/or output is correctly handled. The same is true for activity parameter nodes. Join nodes have more than one input and forks have more than one output. Also decision and merge nodes may have multiple input and output edges respectively. This is why their respective rules are also iterative. Signal and accept signal nodes are a special case since they are only conceptually linked by their signal identifier. Therefore a transition is required between the corresponding processing places. Note that the ActivityEdgeMapping rule is executed after all these rules again in order to cover any remaining required connections for the multiple inputs and outputs case.

3) Interruptible Regions: Figure 5 depicts the only rule of the fifth phase that handles interruptible regions. An interruptible region contains a particular accept signal node such that, when it receives the appropriate signal, any

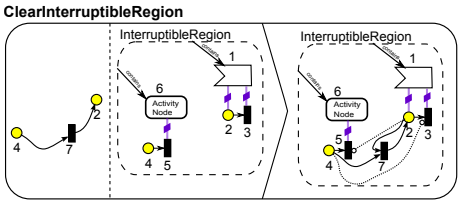

Figure 5. The third set of rules: Interruptible region.

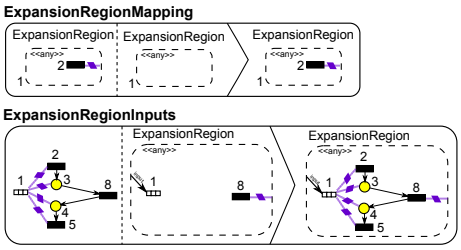

Figure 6. Expansion region mapping.

processing activity node inside the region is interrupted. The accept signal node is then executed and the control follows in the action connected to it. Therefore, when an interrupt signal occurs, tokens in all places corresponding to an AD element in the region must be cleared. The transition labeled 7 in Figure 5 is responsible for consuming all these tokens. The inhibitor arcs are required in order to force this transition to be fired first and then the transition labeled 3 can fire.

4) Expansion Regions: The remaining phases deal with expansion regions. There are three kinds of expansion regions. In an *iterative expansion region*, only one set of inputs (singleton in case of one input expansion node) can be processed by the region at a time. The remaining sets of input are queued in the input expansion node(s). In a *parallel expansion region*, multiple sets of inputs can be processed concurrently in the region. A *stream expansion region* behaves similarly to the parallel one, but it receives a continuous input flow. Furthermore, tokens are accumulated in the output expansion node(s) until all are processed. Only then are they output from the expansion region. In Figure 6, all three kinds of expansion regions are mapped to a transition. This transition is used to control the input flow.

For the iterative case, additional mapping rules are required to satisfy the blocking condition on the input, as shown in Figure 7. For that, an extra transition is added to the output of the expansion region together with a one-token place. The output expansion nodes must also be controlled so their corresponding transitions are fired together. Special care must be taken for activity final and flow final nodes inside an iterative expansion region, since their PN counterpart does not output any token. The token present in their processing place must also enable the output transition.

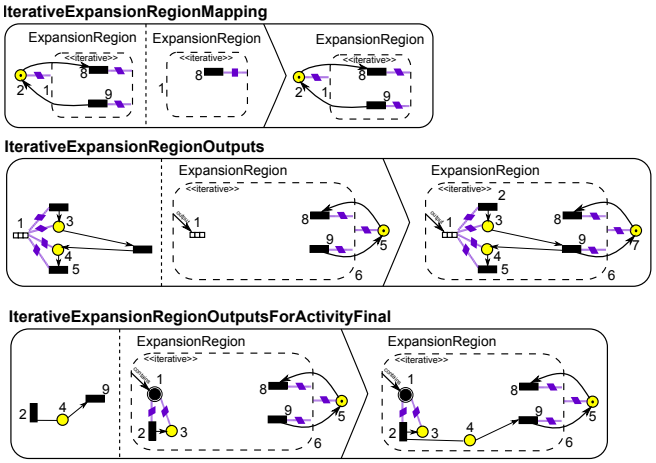

Figure 7. Iterative expansion region mapping.

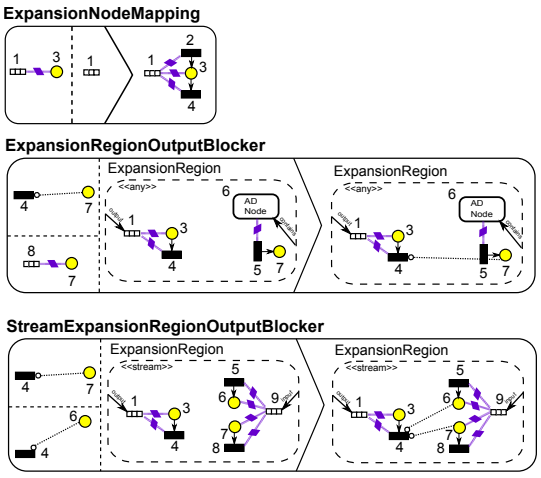

Figure 8. Expansion node mapping and output blockers.

The next phase (ExpansionNodeMapping rule in Figure 8) maps expansion nodes of parallel and stream expansion regions. Note that the NAC of the rule prevents from mapping the ones of iterative expansion regions that were already mapped in the previous phase. In the last phase, the ExpansionRegionOutputBlocker rule in Figure 8 ensures the appropriate output blocking semantics. All expansion nodes must consume all tokens inside them to output. For that, an inhibitor arc is required to prevent the output transition from firing if there are still tokens in any places inside the region. For the stream case, recall that there is a continuous flow of input to the expansion region. Therefore an additional inhibitor arc must make sure that all possible input is processed before sending the output out of the expansion region node.

The presented translation results in PN models where only one transition is fireable at the beginning (corresponding to the initial node). But this does not mean that there will not be concurrent flows, since more tokens may be produced

Table I
SCHEDULING OF THE TRANSLATION RULES

Phase	Rule
1	ActionNodeMapping^F, TimeEventMapping^F, InitialNodeMapping^F, ForkNodeMapping^F, JoinNodeMapping^F, ObjectNodeMapping^F, ActivityParameterNodeMapping^F, DecisionMapping^F, MergeMapping^F, ActivityFinalMapping^F, FlowFinalMapping^F, SignalMapping^F, AcceptSignalMapping^F
2	ActivityEdgeMapping*, ExceptionEdgeMapping*
3	ActivityEdgeBetweenObjectNode&ActionMapping*, JoinInputEdgesMapping*, ForkOutputEdgesMapping*, DecisionInputMapping*, MergeOutputMapping*, MatchSignals*
4	ActivityEdgeMapping*
5	ClearInterruptibleRegion*
6	ExpansionRegionMapping^F
7	ExpansionRegionInputs*
8	IterativeExpansionRegionMapping^F
9	IterativeExpansionRegionOutputs*, IterativeExpansionRegionOutputsForActivityFinal*, IterativeExpansionRegionOutputsForFlowFinal*
10	ExpansionNodeMapping^F
11	ExpansionRegionOutputBlocker*, StreamExpansionRegionOutputBlocker*

from *e.g.,* fork nodes and multiple object nodes. Note that an Activity is not mapped to any PN counterpart. That is because its operational semantics is defined by its enclosed action nodes. The transformation also assumes that edges adjacent to an activity are connected through object nodes only. Activity partitions (or swimlanes) are simply used for visual convenience and do not have any operational semantics.

C. Operational Semantics

We are now able to translate any AD model to a behaviorally equivalent PN model. In order to simulate the AD model, we design a PN simulator in MoTif that defines the operational semantics of the PN model and, by transitivity, the operational semantics of the AD it was translated from. The simulator is extended from [15] to handle inhibitor arcs with a simple rule that makes sure no places are inhibiting a fireable transition. We also adapt the animation rules to highlight an AD element only when tokens are present in its processing place. AToM^3 then automatically takes care of updating the respective concrete syntax.

Figure 9 shows the rules used for the simulation. As in [15], the transformation executes in an infinite loop. The first rule FindTransition (which is a query consisting of solely a LHS) loops over each transition in the model. The transition found by this rule is assigned to a pivot

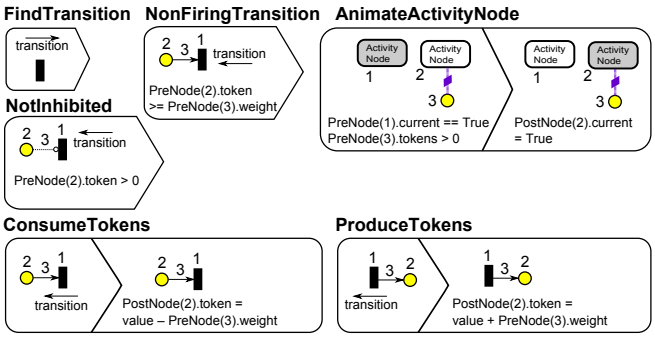

Figure 9. The rules for the simulation of PN and animation of AD.

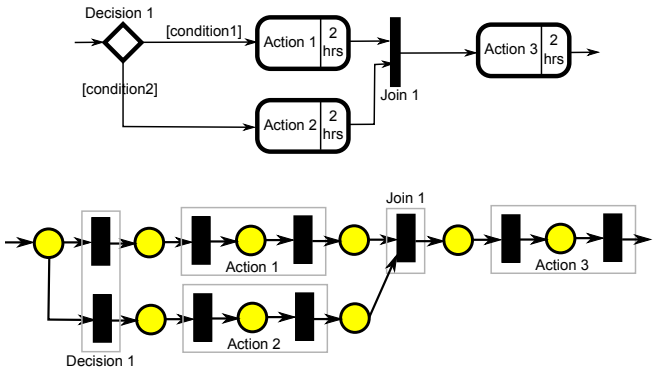

Figure 10. A deadlock scenario in AD detectable by its PN equivalent.

variable `transition` to be referred by subsequent rules. We then must ensure that only firing transitions will be processed. To find enabled transitions, we iterate through all transitions until one has been found that does *not* satisfy the pattern of a *non*-firing transition. This is done by iterating over every arc input to the selected transition and, if the IsNonFiring rule cannot succeed, the transition is fireable. An additional rule NotInhibited ensures there are no places inhibiting the transition. If the selected transition passed the previous two conditions, tokens are transferred along this transition as depicted by rules ConsumeTokens and ProduceTokens. When the tokens are produced in a processing place, only the corresponding AD element is highlighted to animate the AD. AToM^3 then automatically takes care of updating the respective concrete syntax. After that, the first FindTransition rule is applied again recursively, by re-matching the new model looking for a transition given the new marking. This control flow goes on until no more transitions are fireable.

D. Analysis

Verification of AD is now possible by analyzing the underlying PN model. For example reachability and deadlock analysis can be performed as in [7] and in [9]. Figure 10 depicts a syntactically valid AD model although it has an incorrect design: a decision node is misused with a join node since only one branch will ever be executed and never both. A quick analysis of its equivalent PN clearly shows how the deadlock situation will occur.

In our prototype, we have developed a Petri Net Markup Language (PNML) [16] exporter from AToM^3, which is the standard interchange format for PN models in XML. We have used the Pipe [17] tool to analyze our test models to detect deadlock and safety of the PN. In the future, we plan to incorporate static PN analysis techniques in the AD interpreter to reveal properties of the model instantly. This is very relevant in requirement engineering to provide automated support and instantaneous feed-back to the designer of an AD.

III. Extension for Project Management

We now illustrate the benefits of precisely defining the operational semantics of AD with an application in project management. One way is to design a domain-specific language based on AD to model development activities in a software project [2]. This has many advantages over using regular tools mostly based on spreadsheets, such as:

- the relationship between different activities is explicitly represented;
- a central model can be re-used for scheduling, cost estimation, and resource allocation;
- the simulation of the model facilitates debugging and understanding the overview of the project and improves the communication between customers, analysts, and managers.

Alternatively, one could use UML profiles to extend the AD meta-model for project management, but in general UML profiles require more effort for simple extensions in contrast with building a new meta-model which is much more compact [18].

We propose to extend the AD meta-model with artifacts specific to project management. Figure 11 depicts these modifications and additions. Actions are augmented with duration. The time unit is assumed to be in hours. Human resources can be assigned to specific actions with full or partial availability. The use of human and material resources has an associated cost.

These additions allow us to calculate a prediction of the total project cost and time. A static calculation by summing the actions' durations (taking into consideration overlaps) can give an rough estimate of the timespan of the project. However, the simulation of the extended AD model can increase the accuracy of this estimate. That is because, in practice, resource re-allocation is often a reality. The modeler can take these dynamics into consideration by freeing, adding, or replacing human resources and materials at any time during the simulation. This feature is directly available in our prototype since AToM^3 allows a transformation to run continuously or step-wise. Furthermore, when choices have

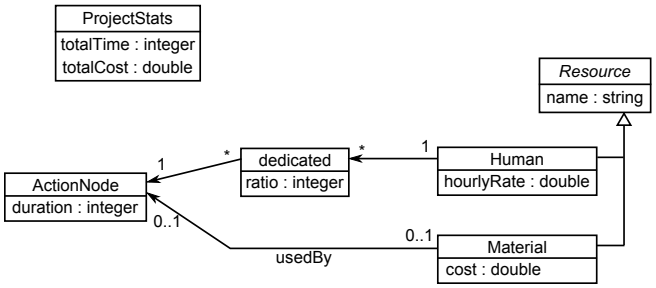

Figure 11. Extension to the AD meta-model.

to be made (*e.g.,* branches, decisions) the transformation can be configured to halt and let the user decide. We have extended the simulator with a rule at the end of the loop that updates the total time and cost of the ProjectStats when the exit transition of an action node is fired. An additional rule enforces that the entry transition of an action is fireable only if it is allocated at least one human resource.

A concrete example of an extended AD model representing a revision process adapted from [19] is shown in Figure 12. Note how for example, action "A1" has human resources and materials attached to it. Figure 13 shows the PN model resulting from the model transformation defined in Section II-B.

IV. Related Work

Since the semantics of UML has deeply changed in version 2.0, we concentrate the comparison with prior works that are relevant to UML 2.x only. There have been numerous studies focusing on the mapping from AD to PN. This is by no means an exhaustive list. However, their main issue was that they only considered basic AD elements, because of the complexity of its more advanced constructs.

Störrle has had a significant amount of contribution in this semantic mapping. He is one of the few to have also considered advanced AD elements. While his works are based on the early stages of UML 2 (2004-2005), we have focused on the recent ISO adoption of UML 2.4.1 which includes slight variations and precisions. In [7], he defined a mapping for basic AD elements to colored PN. The main motivation was to give means to validate AD models by running their equivalent colored PN model. Some preliminary verification was performed through reachability analysis. However, both validation and verification results were not mapped back to the AD model as we do. In [5], Störrle defines a new mapping to procedural PN. This allowed him to reason about the interaction between activity nodes and token objects. He also worked on the semantics of exceptions and interruptible regions in [4]. However, the approach he proposes is not compatible with the description of the ISO reference, since tokens are still present in the interruptible region. In fact, he stated that to fulfill this condition, "flush-arcs would be required which makes things more complex". Our translation

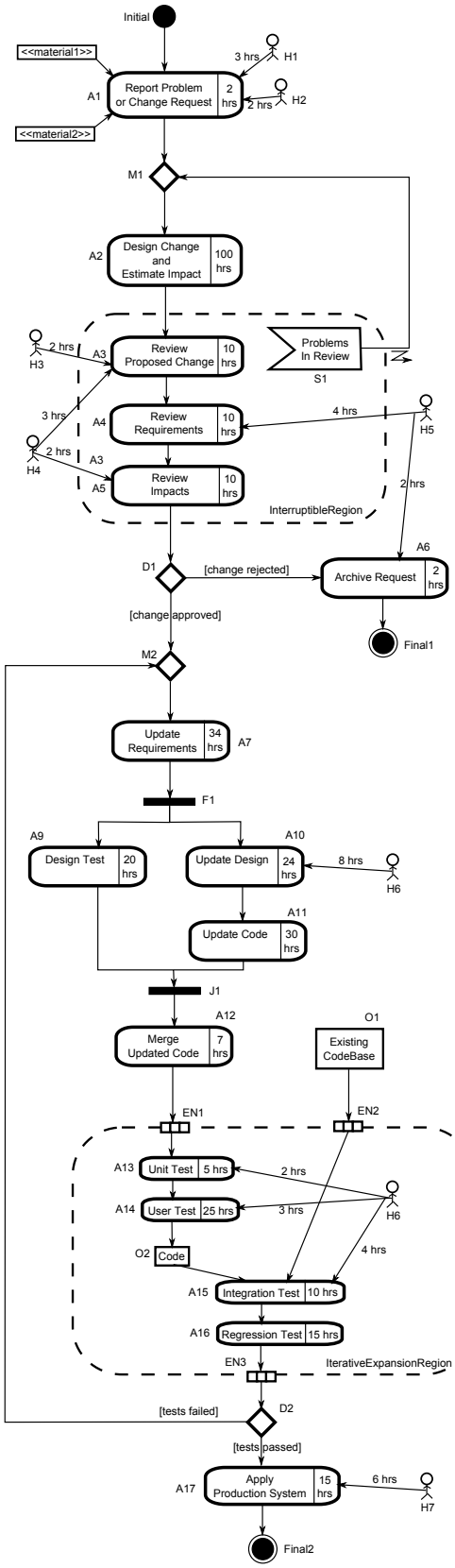

Figure 12. The sample UML activity diagram of a revision process.

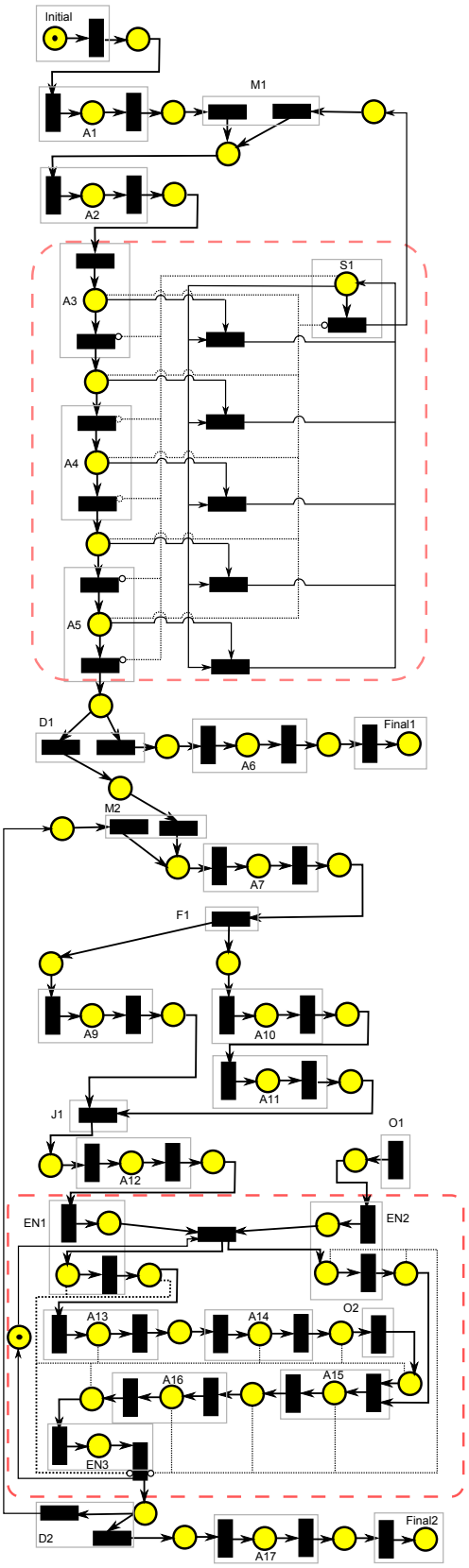

Figure 13. The revision process after mapping.

makes rather simple use of inhibitor arcs. In [6], he defined the semantics of expansion regions in terms of colored PN. His approach is more fine grained than ours: collections of input are split during the process and re-assembled in the appropriate order before they are output.

Han *et al.* [9] again only concentrated on mapping basic AD elements to PN. Actions are mapped to an entry transition and a processing place, which is an alternative representation to ours. They also use reachability analysis to validate structural properties of AD, such as termination.

We are not the first to define the translation using model transformation. Staines [8] used triple graph grammars (TGG), mapping only basic AD elements to PN. Actions were mapped to transitions, which leads to incompatibilities when dealing with expansion regions. One advantage of using TGG is that the transformation is bi-directional. Rafe *et al.* [20] defined the operational semantics of AD as an in-place transformation defined in AGG. However, the simulation expresses only a control flow on basic AD elements. They also used model checking techniques to verify properties of AD expressed in temporal logic.

While we execute AD models in terms of a PN simulator, the work in [21] implemented a dedicated simulator in ARENA to simulate a restricted subset of AD. Specific details of the implementation were not available to us. Finally, Vitolins *et al.* [22] created an AD virtual machine to clearly visualize the movement of tokens in AD models. The simulator is hard-coded in an imperative programming language, whereas ours is declarative and entirely modeled, which has the advantage of being better understood by non-programmers (analysts and project managers) and independent from the platform it is deployed on.

V. Conclusion

We hope that, with the proposed work, stakeholders involved in the requirement engineering process will not use AD as a "convenient notation" but rather as a language with a precisely defined semantics they can understand. For that reason, we translated the full UML 2.4.1 activity diagram adopted by ISO to Petri net, as dictated in the reference manual. We used a rule-based model transformation approach to define the translation as well as its operational semantics. The execution of AD models is done by simulating the underlying PN model. As an application, we have shown how this approach benefits the tasks of project managers in optimally allocating resources and seeing the immediate effects on cost and time estimates.

We are planning to incorporate static PN analysis techniques (inspired from [7], [9] and [20]) and interpret them automatically in terms of the AD model. We will also work on exporting the extended AD models scheduling tools, such as to Microsoft Project (MS Project) [23]. Figure 14 shows a snapshot, from an early prototype, of the Gantt chart corresponding to the AD model of Figure 12. The goal

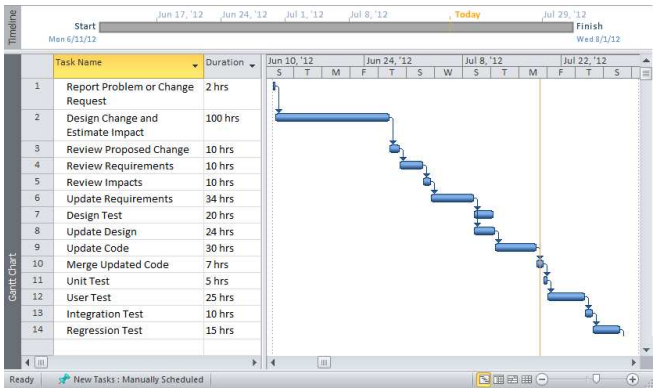

Figure 14. A snapshot of the AD model converted to MS Project.

is to assist project managers in developing plans, assigning resources and tracking activities. We are planning to perform user studies to empirically compare the effect of using AD for project management versus regular tools, such as MS Project.

REFERENCES

[1] B. List and B. Korherr, "An evaluation of conceptual business process modelling languages," in *ACM Symposium on Applied computing*, pp. 1532–1539. ACM, 2006.

[2] A. Gross and J. Doerr, "EPC vs. UML Activity Diagram - Two Experiments Examining their Usefulness for Requirements Engineering," in *IEEE International Requirements Engineering Conference*, pp. 47–56. IEEE Computer Society, 2009.

[3] Object Management Group, *Information technology - Object Management Group Unified Modeling Language, Superstructure ISO/IEC 19505-2*, 2012. [Online]. Available: http://www.omg.org/spec/UML/ISO/19505-2/PDF

[4] H. Störrle, "Semantics of Exceptions in UML 2.0 Activities," Ludwig-Maximilians-Universität München, Tech. Rep. 0403, 2004.

[5] ——, "Semantics of Control-Flow in UML 2.0 Activities," in *Symposium on Visual Languages and Human-Centric Computing*, pp. 235–242. IEEE Computer Society, 2004.

[6] ——, "Structured Nodes in UML 2.0 Activities," *Nordic Journal of Computing*, vol. 11, no. 3, pp. 279–302, 2004.

[7] ——, "Semantics and Verification of Data Flow in UML 2.0 Activities," *Electronic Notes in Theoretical Computer Science*, vol. 127, no. 4, pp. 35–52, 2005.

[8] T. S. Staines, "Intuitive Mapping of UML 2 Activity Diagrams into Fundamental Modeling Concept Petri Net Diagrams and Colored Petri Nets," in *International Conference on the Engineering of Computer Based Systems*, pp. 191–200, 2008.

[9] K. H. Han, S. K. Yoo, and B. Kim, "Qualitative and quantitative analysis of workflows based on the UML activity diagram and Petri net," *WSEAS Transactions on Information Science and Applications*, vol. 6, no. 7, pp. 1249–1258, 2009.

[10] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[11] J. de Lara and H. Vangheluwe, "AToM^3: A Tool for Multi-formalism and Meta-Modelling," in *Fundamental Approaches to Software Engineering*, ser. LNCS, vol. 2306, pp. 174–188. Springer-Verlag, 2002.

[12] E. Syriani, "A Multi-Paradigm Foundation for Model Transformation Language Engineering," Ph.D. Thesis, McGill University, 2011.

[13] E. Syriani and H. Vangheluwe, "A Modular Timed Model Transformation Language," *Journal on Software and Systems Modeling*, vol. 11, pp. 1–28, 2011.

[14] Z. Ujhelyi and G. Bergmann, "Activity Diagrams to Petri Nets - An Eclipse and Viatra Project," http://wiki.eclipse.org/VIATRA2/Activity_Diagrams_to_Petri_Nets, 2011.

[15] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer, "Explicit Transformation Modeling," in *MODELS 2009 Workshops*, ser. LNCS, vol. 6002, pp. 240–255. Springer, 2010.

[16] www.pnml.org.

[17] P. Bonet, C. M. Lladó, R. Puigjaner, and W. J. Knottenbelt, "PIPE v2.5: A Petri Net Tool for Performance Modelling," in *Latin American Conference on Informatic*, 2007.

[18] Ingo Weisemöller and Andy Schürr, "A Comparison of Standard Compliant Ways to Define Domain Specific Languages," in *MoDELS Workshops*, pp. 47–58. Springer, 2008.

[19] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering Using UML, Patterns, And Java*, 3rd ed. Pearson, 2010.

[20] V. Rafe and A. Rahmani, "Formal Analysis of Workflows Using UML 2.0 Activities and Graph Transformation Systems," in *Theoretical Aspects of Computing*, ser. LNCS, vol. 5160, pp. 305–318. Springer, 2008.

[21] A. Teilans, A. Kleins, Y. Merkuryev, and A. Grinbergs, "Design of UML models and their simulation using ARENA," *WSEAS Transactions on Computer Research*, vol. 3, no. 1, pp. 67–73, 2008.

[22] V. Vitolins and A. Kalnins, "Semantics of UML 2.0 Activity Diagram for Business Modeling by Means of Virtual Machine," in *International EDOC Enterprise Computing Conference*, pp. 181–194. IEEE Computer Society, 2005.

[23] http://www.microsoft.com/project/en-us/product-information.aspx.