A Triple Graph Grammar (TGG) Approach for Mapping UML 2 Activities into Petri Nets

Anthony Spiteri Staines
Dept. of Computer Information Systems
University of Malta
Msida MSD 2080
Malta
toni_staines@yahoo.com

Abstract: - Model-to-Model mapping offers several advantages over relational mapping. In model-to-model mapping an active correspondence is kept between two pairs of models. This is facilitated if visual models are used. UML 2 activities are based on Petri net like semantics and substantial literature exists explaining the conversion process. This paper explains how UML 2 activities can be formally mapped into Petri nets or Petri net semantics from a theoretical and practical and operational point of view adding on previous work of Triple Graph Grammars (TGGs). UML activity constructs have been classified and identified for creating a basic set of TGG rules. The concepts presented can easily be developed further and even extended to other visual models or notations.

Key-Words: - Triple Graph Grammars, Petri Nets, Activities, UML2

1 Introduction
At present UML 2 activity diagrams are structured visual modeling notations used for describing different types of behavior found in information systems [2], [17]. Some practical uses of activity models are for i) web processing, ii) web service composition, iii) business process modeling, iv) workflow modeling, v) systems integration, vi) task management and vii) low level tasks like software operations. UML 2 activities are suitable for modeling the diverse requirements of many traditional scenarios. Activities provide for visual modeling that is easily understood. Activity processes can easily be abstracted into activity models. High level models are suitable for transformation into executable processes and different languages. Activity models are not a proper formalism and need proper verification and validation. Transforming activities into Petri nets or Petri net classes seems to be the best solution. This is evidenced from previous work [7]-[9],[18].

2 Related Works
Different research exists evidencing the need to support UML notations using Petri net models. Some examples are found in [1], [3]-[9], [13]-[16]. One method of transforming UML use case constructs to colored Petri nets (CPN) is based on multi layers [1]. Use cases are the starting point for activity modelling, where no proper formalisms have been used. In [2] a UML 2 activity model for an online multi role playing game is transformed into a special type of Petri net (PEPA net) and analyzed. The transformation process is again informal.

A well structured, semi formal method is presented to translate activities into LGSPNs (labeled generalized stochastic Petri nets) in [3]. These are very useful for performance analysis. Unfortunately the approach seems to be a relational one. Case tools found in the LaQuSo project [5] are used to transform basic activity diagrams into simple Petri nets. HLTPNs (higher level timed Petri nets) have also been indicated for supporting and formalizing the UML.

In [7]-[9] it is explained how activity semantics and constructs are classified and translated into Petri net semantics. The preferred Petri nets class indicated are colored Petri nets or higher order nets.

The UML can be formalized using Petri net like semantics. A CPN based formalization of the UML is presented in [6]. Transforming UML 2 activities into a Petri net semantics has been formalized in [5] and [8]. In [8] a semantic function is described and given. This converts an activity diagram <activity node, activity edge> into a CPN. Practically all these approaches are more about relational issues rather than operational. Petri nets are also found in Fundamental modelling concepts and apply to activity modelling in this context. According to [11] TGGs are suitable for expressing UML activity
3 Motivation for Activity Mapping

The motivation for transforming activities into Petri nets is that UML 2 activities are based on Petri Net like semantics according to the UML 2 superstructure specification. Activities have a higher level of abstraction. Activities share common properties with Petri nets. This is not properly explained in the UML specification. Different approaches to transforming activities into Petri net classes have been suggested. Some are informal, others are semi formal or completely formal. Some of these approaches are quite complex. E.g. a transformation function can be used. These approaches still do not explain the actual transformation process and normally the transformation is too cumbersome to use. To solve this issue a model to model mapping using a Triple Graph Grammar approach is being proposed. Model transformation and visual model mapping has become increasingly popular over the years. This is evident from the OMG approach where QVT (query, view, transform) is used to support UML and also with work related to ATL (Atlas transformation language).

Model-to-model mapping offers several advantages over other approaches. Transformation with TGGs is not only relational but also operational as indicated in [12].

3 A Triple Graph Grammar Solution

3.1 Triple Graph Grammars

Triple graph grammars (TGGs) have been around for several years. They are useful techniques for mapping two different types of graphical models sharing some similar properties. With TGGs it is possible to i) define and ii) declare bi-directional transformations. Relationships between the different models need to be established. A model can be transformed into another model and correspondence is computed incrementally. All changes are recorded and changes can be synchronized. These approaches use similar concepts to those found in TGGs. TGGs have been well researched and documented. There are many different examples of TGGs uses in literature. TGGs describe the dynamic evolution of different models.

TGGs are based on graph grammars. In simple terms graph grammars are visual rules explaining how a graph or part of it, is to be modified according to certain conditions. E.g. nodes and edges are added or removed accordingly. Normally a graph grammar rule has a right hand side and a left hand side that describes the rule.

TGG rules are similar to graph grammar rules with the exception that in TGGs there are three lanes or domains represented [11],[12]. The left side represents the model, the middle represents the correspondence and the right lane represents the transformed model or result. Correspondency mapping is used to link both models from different domains. This is shown in fig. 1. If one model changes, the corresponding model is updated via the application of rules. The rules can be applied: never, once or a number of times.

![UML Activity to Petri net TGG Mapping](image)

Fig. 1 UML Activity to Petri net TGG Mapping

3.2 Classifying UML2 Activity Notations

To create TGG rules the starting point is to comprehend the underlying model relationships and abstract them into a TGG rules for a particular relationship. The conversion rules are defined in terms of TGGs. They explain how a typical activity can be converted into a Petri net. The rules cover all the basic constructs of UML 2 activities. UML 2 activities are divided into seven main types which are :i) fundamental, ii) basic , iii) intermediate, iv) complete, v) structured , vi) complete structured and vii) extra structured. Structured activities focus on traditional programming, whilst fundamental and basic activities have a level of abstraction making them ideal for high level business process modelling. Given several classes of activities according to the UML superstructure specification it is possible to identify different constructs being used, but the most important constructs are used repeatedly in the different classes. The constructs considered are those found in intermediate activities which inherit from basic activities. For this work we
have not specified to which classes these constructs belong, but they are some of the most common.

Technically speaking activities are composed of i) nodes and ii) edges. A special edge is used for connecting parts of object nodes. These are ignored here. There are other constructs like signal nodes and edges which can be similarly treated to object nodes.

Fig. 2 explains the classification of UML 2 activities into nodes and edges as required for transforming them into Petri nets. This is based on previous work presented in [7]-[9], [18]. The actual corresponding Petri net constructs are shown in fig. 3 and 4.

Activity edges can be decomposed into control flow edges. Control flow edges are composed of i) normal control flows and ii) exception control flows. For exception control flows or edges there are i) control node to control node special edges, ii) edge to control node and iii) control node to edge special edges. Control node-to-control node special edges are composed of: i) merge-to-merge nodes, ii) start-to-merge nodes, iii) merge-to-activity final nodes and iv) merge-to-flow final nodes. Edge to control node special edges are composed of: i) edge-to-activity final, ii) edge-to-flow final and control node to edge special edges have start-to-edge. These exceptions require special treatment unlike normal edges. A normal activity edge or normal control flow maps into a Petri net i) input arc, ii) connected to a place and a iii) output arc from the place. A normal action node or executable node translates or maps into a transition. Exception activity edges are explained in fig. 3.

Activity nodes are classified into i) action nodes, ii) control nodes, iii) object nodes and iv) executable nodes.  Action nodes and executable nodes convert to Petri net transition types. Control nodes and object nodes convert to Petri net place types. Control nodes can be sub classified into i) fork nodes , ii) merge node, iii) initial node, iv) final node and v) decision node.

Control nodes can be treated similarly and do not constitute an exception. Fork and join nodes are an exception to this. A fork or join node is treated as an executable node and converts into a Petri net transition. Activity nodes are explained in fig. 4.

For conversion the procedure that follows is to start translating the activity model from the starting node visiting every edge and node in the activity diagram applying each TGG rule in sequence. All nodes and edges need to be covered. TGG rules need to be constructed for normal and exception behavior that has been explained above.

### 3.3 Triple Graph Grammar Mapping Rules

For simplification purposes addition or insertion rules shown by ++ are used to illustrate the mapping.
concept [12]. There are three domains represented in each rule. This is the i) activity model, ii) correspondence and iii) Petri net. To simplify the rules the Petri net domain node is not always shown. A total of six TGG rules are created from the classification of the UML activity notations.

<table>
<thead>
<tr>
<th>Activity Edges</th>
<th>Corresponding Petri Net Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name, text or expression</td>
<td>Name, text or expression</td>
</tr>
<tr>
<td>( S_1 \rightarrow n \rightarrow S_2 )</td>
<td>( n \rightarrow S_1 \rightarrow S_2 )</td>
</tr>
</tbody>
</table>

Fig. 3 Activity Edges and corresponding Petri net constructs

<table>
<thead>
<tr>
<th>Activity Edges Exceptions 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action Nodes</td>
</tr>
<tr>
<td>Object Nodes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Activity Edges Exceptions 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Nodes</td>
</tr>
<tr>
<td>Fork or Join Nodes</td>
</tr>
</tbody>
</table>

Fig. 4 Activity Nodes and corresponding Petri net constructs

The rules are: Rule1 Add a New Control Node. This rule excludes fork or join nodes. Rule2 Add a New Executable Node, Action Node or Fork/Join Node. Rule3 Insert a Normal Activity Edge between existing Action Nodes. Rule4 Insert an Exception Activity Edge between Two Control Nodes. Rule5 Insert an Exception Activity Edge for Executable to Control Node. Rule6 Insert an Exception Activity Edge for Initial Node to Executable Node. These are shown in fig. 5-10.

Fig. 5 TGG Rule 1: Adding a New Control Node (excluding fork or join nodes)

Fig. 6 TGG Rule 2: Add a New Executable Node, Action Node or Fork/Join Nodes

Fig. 7 TGG Rule 3: Insert a Normal Activity Edge

Fig. 7 describes the generalized process of inserting a normal activity edge between action nodes and executable nodes. A fork or join construct in the activity model is treated as one of these. For Fig. 9 again the executable node is a generalization for action node, fork or join nodes. In all the diagrams the shading shows the insertion part.
4 A Simple Example

A toy example similar to that found in the UML 2 superstructure specification [17] is used to illustrate the mapping process. Fig. 11 shows a basic activity diagram for processing a simple customer order. The TGG rules can be applied to obtain the Petri net shown in fig. 12.

4 Conclusion and Future Work

This work presented has dealt with the formal mapping of basic UML 2 activity diagrams into Petri nets. It has explained how this can be successfully achieved using TGGs. The TGG rules just specify the correspondence mapping of activities and Petri nets graphically using an abstract notation. These rules can be made operational in different ways and applications.

TGGs are useful for formal mapping of two similar visual notations. Many steps and rules are required and have to be repeated several times for transformation. I.e. the more complex the model the rules must be applied a greater number of times. I.e. just adding a start node requires two rules. Buffer nodes, queues and other complex constructs etc. require special attention.

The TGG rules presented are a simplification and generalization of what needs to be done. Only addition rules denoted by ++ have been used. Actually deletion rules that have not been presented might need to be considered. For mapping the control nodes edges and edge insertion see fig. 5-7.
these are generalized or generic rules for different activity constructs. From these rules it is possible to create specific rules for every different control node type. E.g. for just adding merge node, final node, initial node three separate rules need to be created. This means that over ten rules are required. The conversion will become complex to manage. A CASE tool would help with this.

The TGG rule drawings presented in this paper just give a rough indication of what needs to be done. They definitely can be improved to introduce more detail. They have been constructed just to indicate the most salient points. They also can be colored appropriately to indicate what is being inserted. Other transformational concepts like model transformation model integration, model synchronization node reusability, constraints need to be considered as is done in [12]. Attributes and other advanced concepts can be considered. Finally for mapping the two models other work has to be done and applied.

References: