International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:6, 2008

Formal Modeling and Verification of Software Models

Siamak Rasulzadeh

***Abstract*—**Graph transformation has recently become more and more popular as a general visual modeling language to formally state the dynamic semantics of the designed models. Especially, it is a very natural formalism for languages which basically are graph (e.g. UML). Using this technique, we present a highly understandable yet precise approach to formally model and analyze the behavioral semantics of UML 2.0 Activity diagrams. In our proposal, AGG is used to design Activities, then using our previous approach to model checking graph transformation systems, designers can verify and analyze designed Activity diagrams by checking the interesting properties as combination of graph rules and LTL (Linear Temporal Logic) formulas on the Activities.

***Keywords*—**UML 2.0 Activity, Verification, Model Checking, Graph Transformation, Dynamic Semantics.

I. Introduction

UML Activity diagrams are suitable means to model dynamic parts of a system. They allow modeling of complex and large processes or specifying workflows [1]. They can be used to model the behavior of a system or to specify the global behavior of a service-oriented architecture [2]. Oftentimes, however, modeling must be complemented with suitable analysis capabilities to let the user understand whether the designed model fulfills the stated requirements. To have a precise analysis in an automated way, design models like Activities should be stated with a formal language (i.e. a language with a precise semantics).

Since the past decade, Unified Modeling Language (UML) has been a standard modeling language to express models in a software development process. The major drawback of UML is that it only defines syntax for modeling without a precise formal semantics. Formal methods are crucial in automated software engineering. But the problem of formal methods is that they are difficult to be understood by designers because there is a complex mathematics behind them. Hence, our aim is to implement a precise semantics –based on UML 2.0 specification [3]-yet easily understandable for UML 2.0 Activities using graph transformation systems [4,5].

Graph transformation has recently become more and more popular as a general formal modeling language. Many of the artifacts which software engineers are used to deal with are nothing but suitable annotated graphs. Software architectures, class diagrams, and version histories are only a few well-known examples in which graphs have proven their usefulness in everyday software engineering. These models, and many others, can easily be described by means of suitable graph transformation systems to formalize their syntax and define the formal semantics of used notations [6]. Hence, graph transformation is a natural formalism for languages which basically are graphs and this motivates us to choose graph transformation as a semantic background for modeling Activities.

To analyze Activities –modeled by graph transformation system-we use model checking. For doing so, based on our defined semantics, the transition system must be generated. In the generated transition system states are graphs representing the current state of the Activity. Then it is possible to check specified properties of the model (e.g. via temporal logics interpreted on the transition system). To implement semantics of UML 2.0 Activity diagrams, we use AGG[1] toolset [7]. AGG supports attributed typed graphs and layered graph transformation systems. It is also possible to define desired constraints using atomic constraints in AGG. As AGG cannot generate transition systems, we use our previous approach to generate transition systems and to do model checking [8]. We translate graph transformation systems designed in AGG to BIR (Bandera Intermediate Language) –the input language of Bogor[2] model checker [9]-. Bogor generates the transition system and checks desired properties stated by LTL. This translation is done automatically and designers can use this approach without any knowledge about the BIR or Bogor.

As it was mentioned before, one of the main application areas of the Activities is workflow modeling. Hence, we use our proposed semantics of UML 2.0 Activity diagrams for modeling workflows. To verify the correctness of workflows, we consider several crucial properties of Activities modeling workflows. We describe how our proposed semantics can be used automatically to verify workflows. In contrast to previous approaches [10,11], our proposed semantics supports concepts defined in UML 2.0 Activities (e.g. Petri-like semantics and traverse-to-completion). Furthermore, our approach can cover more elements of Activities (e.g. exception handling and events) for modeling than [12]. Also, it has more flexibility to check user defined properties on the Activities.

Siamak Rasulzadeh is with Islamic Azad University, Naragh Branch, Iran (e-mail: S_rasulzadeh@yahoo.com).

[1] http://tfs.cs.tu-berlin.de/agg/
[2] http://bogor.projects.cis.ksu.edu/

The paper is organized as follows. Section II surveys the related works. Section III describes our solution to define a formal semantics for Activities. Section IV shows our approach to verify modeled Activity diagrams and section V concludes the paper.

II. Related Work

There is much research done about definition of formal semantics for Activity diagrams using different formal languages. In [13], Hausmann defines a concept named Dynamic Meta Modeling (DMM) using graph transformation systems. He extends the traditional graph rules by defining a new concept named "rule invocation". In DMM, there are two kinds of rules: big-step and small-step rules. Big-step rules act as traditional rules but small-step rules should be invoked by big-step rules. Hausmann then defines semantics for Activity diagrams using concept of DMM. Engels et al. [14] use DMM and semantics defined by Hausmann for modeling and verification of workflows. For verification, they use GROOVE [15], but as GROOVE does not support rule invocation, they change the rules to be verifiable by GROOVE. They check deadlock freeness and action reachability properties on the modeled workflows. In contrast to this work, our approach has more flexibility to support user defined properties. Furthermore, our approach has the ability to support data flow; also event and exception modeling can be supported. Additionally, the extension defined by Hausmann (small/big step rules and rule invocation) can not be modeled directly in existing graph transformation tools; hence it is not so easy for designers to use this approach.

Störrle et al. [16] use Petri nets as the semantic background for the UML 2.0 Activities. They examine Activities as described in the UML version 2.0 standard by defining denotational semantics. It covers basic control flow and data flow, expansion nodes and exception handling. They show that some of the constructs proposed in the standard are not so easily formalized by Petri nets. Due to the traverse-to-completion semantics in UML 2.0, they conclude that it is not possible to use Petri nets for this purpose.

Eshuis [17] defines a statechart-like semantics for UML 1.5 Activity diagrams. He defines a property called 'strong fairness' (the model should not have any infinite loops) to verify functional requirements of the model. This approach uses NuSMV model checker [18] to check the strong fairness property stated in an LTL expression. This approach and others [11,19] do not treat UML 2.0 Activities, but its 1.5 predecessor.

Baldan et al. [20] use hypergraphs to show the behavior of a model (instance graph) by using UML Activities (rather than to define semantics for Activities). They use instance graph to show the static model of a system, then by defining a rule for each Action in the Activity and using synchronized hypergraph rewriting, they control the application of the rules. They present a variant of monadic second-order logic to verify hypergraphs. But they do not introduce any tools to implement their ideas. Furthermore, they do not use semantics defined by UML 2.0 (e.g. token flow) to implement their proposal.

III. Implementing the Semantics

To ease modeling of workflows we only use a subset of UML 2.0 Activities, since using this subset suffices to model many types of workflows. Our approach focuses mainly on control flow perspective; but it also can handle data flow. The parts of Activities which we consider for workflow modeling are shown in Fig. 1. Before we present our defined semantics for workflow modeling, we need to show its basic idea. According to the UML 2.0 specification [3] "Activities have a Petri net-like semantics", i.e., the semantics is based on token flow. When an Activity is executed, the *Init* node starts the flow of token. Then based on our defined rules this token is routed through the Activity.

Taking the semantics described above into account, we need to define an accurate syntax for the models under design. For doing so, we define some constraints on the models. These constraints are as following:

1. Each Activity diagram must have exactly one *Init* node and one *Final* node.
2. *Init* node has no incoming edges and *Final* node has no outgoing edges.
3. Each *Fork* and *Decision* node should have exactly two outgoing edges.
4. Each *Merge, Action*, *Object*, *Init* and *Join* node must have exactly one outgoing edge.
5. The source and target node of each edge should not be identical. (There must not be any self-edge in the graphs.)
6. Each *Final*, *Object*, *Action*, *Fork* and *Decision* node should have only one incoming edge.
7. Each *Join* and *Merge* node should have exactly two incoming edges.
8. Each *Action* node can have some outgoing edges to some different *ExceptionHandler* node and each *ExceptionHandler* node should have exactly one outgoing edge to an *Action* node (this kind of edges is different from other edges).

Notice that in practice, constraints 1,3 and 7 do not restrict the modeler: more than one Init node can be modeled equivalently by one Init node and one or more Fork node(s). (Final and Join nodes accordingly). Fork (or Decision) nodes with more than two outgoing edges can be modeled equivalently by cascading two or more Fork (Decision) nodes (we use the same way for Join and Merge nodes). We have proposed these constraints to have models with precise syntax and it is possible to draw many of UML 2.0 Activities using these constructs[3].

The class diagram shown in Fig. 1 represents a portion of UML 2.0 Activity diagrams' metamodel [21]. This metamodel can be formally considered as an attributed typed graph. The

[3] We do not consider labels or guards on the edges because it has not any effect on our approach for verification.

abstract syntax of a modeling language is defined by a metamodel and it can be represented formally as a type graph. Since UML 2.0 specification stipulates that activities "use a Petri-like semantics" [21], therefore; to show tokens, we add an attribute to Action node, named "token" of type "boolean". Fig. 2 shows the enhanced metamodel as a type graph for Activity diagrams.

Fig. 2 shows the proposed type graph based on the enhanced metamodel and listed constraints. This type graph and other parts of proposed graph transformation system are designed in AGG toolset. AGG automatically checks that each host graph (i.e. Activity diagram) and rules are consistent with its type graph and other constraints. Modeling an Activity in AGG ensures us that it is syntactically consistent with type graph and other constraints.

The proposed type graph comprises one abstract type (i.e. *Node)*, the star (*) sign on the top right corner shows the multiplicity of these nodes in the models. Other nodes (except *Exception*) have inherited this abstract type. Using inheritance, all the other nodes (except *Exception*) have the associations with the specified multiplicities to node type "*Edge*". Based on UML 2.0 specification, only *Action* and *Object* nodes can hold the token, while control nodes and edges can not. Preventing control nodes and edges from holding tokens ensures that tokens do not get "stuck" when alternative paths are open [22].Taking this semantics into account, we enhanced *Action* nodes with a "Token" attribute. For *Object* nodes there are two kinds of these attributes. Since *Object* nodes can hold more than one token at a time and each object node specifies the maximum number of tokens it can hold [22], we used "UB" to show the upper bound of tokens. "CV" shows the current value of tokens. Before the execution of the Activity, the value of "CV" is zero and it means that the Object node does not carry any token. Greater values for "CV" show the number of tokens in an Object node and thus we do not need "token" attribute for Object nodes. Node type "*Exception*" is used to support exception handling. Each *Action* can raise one or more exceptions and each exception is associated with a handler (it is specified with an *Action* node). The type graph has a node type "Key". This node type has a boolean attribute "flag". Each Activity has exactly one node of type "key" and the initial value for "flag" is false. We use this node to control the execution of the Activity at the beginning and at the end (by some graph rules). Each node type "Edge" points to the "token" which it wants to route (using the "points" association).

This type graph does not satisfy all the above constraints. Therefore, we need some more constraints besides this type graph. For example, based on this type graph, it is possible to have host graphs (Activity diagrams) with some *Edge* without the source or target nodes. In AGG, using *atomic graph constraint* and *formula constraint*, we can define desired constraints on the model. As an example, consider Fig. 3. It consists of three *atomic graph constraints* which have been described by three rules. First one depicts each *Edge* must have exactly one source and one target node (and the source and target are not identical). The second constraint states each node can not have two outgoing edges to a node of type "*Edge*" (we will use the negation of this constraint in the *formula constraint*). The last constraint says each *Fork* node must have exactly two outgoing *Edges*. Then the *formula constraint:* (1 && (! 2) && 3) states that all the host graphs (Activities) should follow these three constraints.

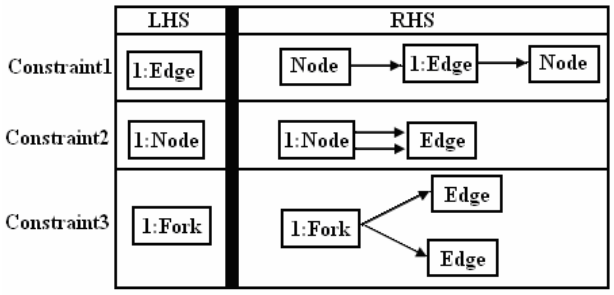

Fig. 3 Three atomic graph constraints

After adding other constraints to the graph transformation system, the metamodel of the proposed formal semantics is completed. Now, we can model workflows (or Activities) directly as a host graph in graph transformation system. Fig. 4 shows a sample workflow modeled by an Activity diagram [3]. The dashed region shows the area that the exception "*Cancel Order Request"* can be raised. All *Action* nodes in this area can raise this exception. We will use this Activity diagram as a running example for the rest of this paper. It describes the processing of orders in a company. The meaning of this diagram is supposed as follows:

When an order arrives, it might be accepted or rejected (the *Decision* node with two guards shows this fact). In the case of acceptance, Action *Fill Order* must be done. Then, to speed up the process, two Actions *Ship Order* and *Send Invoice* are performed in parallel. When these two actions are terminated (or the *order* has been rejected), *Order Close* Action is performed. Finally either by reaching the *Final* node or raising the exception *Cancel Order Request*, the process is terminated.

In our approach, this diagram must be modeled in AGG based on the type graph (metamodel) of Fig. 3. For the lack of space we cannot show the equivalent host graph for Activity of Fig. 4. Notice that we have modeled workflows directly as a host graph (Activity) in AGG, but it is possible to draw the Activity in a desired UML editor. Then using a one to one mapping between UML and AGG constructs, we can implement a transformer.

Most of the constraints formulated above put restrictions on the syntax of Activities rather than on their semantics. Since syntax restrictions can be automatically verified by AGG, their verifications will not be discussed further. The dynamic semantics are more important. To verify them, at first we need a formal semantics of the behavior of Activity diagrams. Hence, the next step is designing the dynamic semantics of the model by defining graph transformation rules. The proposed rules show the token flow in the host graphs. To define these rules we consider following definitions for token flow:

1. In each workflow, at first, there is no token, i.e. the *Edge* nodes do not point to any node. The token flow will be started by *Init* node as soon as the Activity is executed.
2. Tokens can not get stuck on nodes, it means as soon as there is a suitable way, the token should be routed. This is compliance with the UML specification which states the traverse-to-completion semantics for tokens.
3. The flow of tokens will be terminated as soon as a *token* arrives to *Final* node.

Based on these definitions and the desired behavior of Activity diagrams, we have proposed 26 graph transformation rules as dynamic semantics for Activities. Due to the lack of space, we can not explain all of them here, but we briefly describe some of them.

We have implemented the token flow semantics in a simple way: as soon as a token arrives at the incoming *Edge*(s) of a node (in this case the *Edge* points to *Action* or *Object* node holding the token), this node (based on its defined semantics) will offer the token to its following *Edge*(s) till it reached to incoming *Edge* of another *Action* or *Object* and in this case, the holding *Action* or *Object* node will really pass the token (and this is the traverse to completion).

Fig. 5 shows a rule implementing the semantics of the *Init* node. NAC (Negative Application Condition) and LHS (Left Hand Side) describe the preconditions, while RHS (Right Hand Side) shows the post-conditions of the rule. We have used notation defined by AGG to show the rules. The rule shown in Fig. 5 depicts that if *flag* attribute of node *Key* is false and the outgoing *Edge* of *Init* node does not point to *Key* node, then the *Edge* must point to the *Key* node (in this case *Key* node plays the role of a token). In fact, this rule shows the starting point for the execution of the Activity. Notice that at the beginning of the execution, there is no existing token in the model. Hence, the *Key* node will play the role of a token till the token arrives at an *Action* or *Object* node. We use the *Key* node (while its *flag* is false) in the LHS of all designed rules.

When a token arrives at the *Final* node, the value of flag will be changed to true. Thus the flow of tokens will be terminated because no more rules can be applied on the Activity. For the lack of space, the rule implementing semantics of *Final* node is not discussed here.

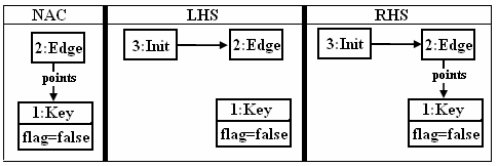

Fig. 5 The proposed rule as the semantics of Init node

Fig. 6 shows two other rules implementing portions of the semantics of *Object* nodes[4]. Rule (A) of Fig. 6 describes a case when the incoming *Edge* of an *Object* node points to a token. The LHS of this rule has an *Attribute Condition*[5]*: AC=x>y* to state that this rule can only be applied on the Activity if the current number of holding tokens by *Object* node is less than the defined Upper Bound for that *Object* node. This is in compliance with the UML 2.0 specification which states that when the number of tokens in an *Object* node reaches its upper bound, it cannot accept any more tokens [22]. In the RHS of this rule, the current number of tokens is increased by one (to show the acceptance of the token by *Object* node) and the incoming *Edge* of the *Object* node does not further point to the token. To show the token in this rule we use An *Action* node, notice that it is possible to use the *Key* or *Object* node instead of the *Action* node. Thus, we need two similar rules for those cases (which are not shown). As it is shown, the token attribute of the *Action* node is changed to false, when the token is accepted by *Object* node. This is compliant with the traverse to completion semantics. The NAC of this rule depicts that this rule can be applied only if there is no other *Edge*(s) in the model which points to it. Some nodes like *Forks* can make copies of the token; hence the token can be false when all the copies have been routed and accepted by destination *Object* or *Action* nodes. For the cases that there are more than one pointing *Edge* to the *Action*/*Object* node, the *Object*/*Action* node in the RHS just receive the token and the token in the LHS remains without changes (we implement this semantic by other rules).

Rule (B) of Fig. 6 shows the case where an *Object* node holding one or more tokens (i.e. AC=y>0) offers a token to its outgoing *Edge*. The NAC of this rule says that the outgoing *Edge* must not point to the token. The RHS of this rule shows that the outgoing *Edge* will point to the *Object* node (i.e. the *Edge* is carrying the token) and the Current Value of tokens remains as the same as LHS. This value will be decreased by one, when another *Action* or *Object* node accepts it in the future (by some other rules like rule (A)).

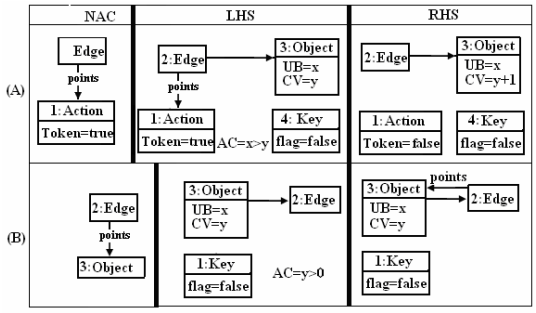

Fig. 6 Two example rules implementing portions of the semantics for Object nodes

Applying all enabled rules to initial state (the Activity diagram before applying rules) will result in a transition system. This resulting transition system represents the complete behavior of the Activity under consideration. It will be the basis for analysis of the Activity, using model checking. In the next section, we show our approach to verify an Activity using its transition system.

[4] In this article we focused mostly on control flow, hence the implemented semantics for Object nodes is based on control flow as opposed to data flow. But, our approach has the flexibility to support data flow.

[5] Attribute Condition (AC) is one of the useful facilities provided by AGG

IV. Verification and Validation

To analyze designed Activities we use our previous approach to verify graph transformation systems [8]. In the cases where designers are expert in graph transformation, they can model workflows directly by graph transformation (rather than UML). In the cases where designers are not familiar with graph transformation, they can model workflows by UML Activities[6] (rather than graph transformation), in that case, they do not need to define any rules for verification, because we have designed some fixed properties to be used by designers. It means designers only model the workflow, and the verification is done automatically via our fixed designed properties (without intervention of designers). In addition, it is possible to define new properties by expert designers. Using rules to state properties has this advantage that designers do not need to learn any other formalism, and they can state properties by the same formalism that they model the system (i.e. graph transformation).

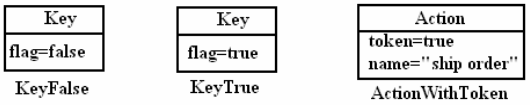

Fig. 7 Rules *KeyFalse*, *KeyTrue* and *ActionWithToken*

First, recall from section III, that for an Activity to be supposed sound, a token must finally arrive at the *Final* node. It means the Activity must be deadlock free. To verify this property we should check that for all possible executions of the Activity, *Final* node is reachable. To state this property, we have designed two rules: *KeyFalse* and *KeyTrue*. Fig. 7 shows these rules. They have no NAC and their LHS and RHS are identical. If a property rule matches a state, we know that the preconditions of that rule hold within the state. The only precondition of *KeyFalse* is that the token has not arrived at the *Final* node (in contrast to *KeyTrue*), notice that when the token arrives at the *Final* node, the flag is changed to true. Hence, we can state this property using the following LTL expression: $\Box(\ KeyFalse \rightarrow \Diamond(\ KeyTrue))$, where symbol "□" means *always*, "◊" means *finally* and symbol "→" shows the *implication*. The result of checking this property on the transition system is true if in every possible execution of the Activity, there is a state in a path in which *KeyFalse* is satisfied and then eventually there is a state in the postfix of that path, in which the token arrives to the *Final* node (i.e. *KeyTrue* is satisfied). It means that the token must always arrive to the *Final* node. As an example where this property is satisfied by the Activity, consider Activity of Fig. 4. In this Activity token always arrive to the *Final* node. But as an example where the mentioned property is not satisfied, consider the Activity of Fig. 8 (a). This Activity shows an Activity which contains a deadlock. In this diagram, there is a *Join* node immediately after *Init* node and it prevents token to be propagated from *Init* node. Therefore, token never reaches to the *Final* node and it causes a deadlock. Hence, the mentioned property is never satisfied for this Activity.

Another property which should hold for a sound workflow is that there must not be any useless work (Action) in it, i.e. for each *Action* node there should be at least one execution in which the token arrives to that node. In other words, each *Action* in a sound workflow must be reachable. We can state this property as the following: each specified Action must be reachable, it means we should design a rule for each Action (using its name) to check this property, Fig. 7 shows a rule (*ActionWithToken*) to state that the Action *"Ship Order"* in the Activity of Fig. 4 is reachable. The following LTL expression states this property: $\neg\Box(\neg$*ActionWithToken*), this property is satisfied on the Activity of Fig. 4. This LTL expression means that there must be an execution of the Activity which the specified *Action* node (*"Ship Order"* in this case) has the token (i.e. it is reachable).

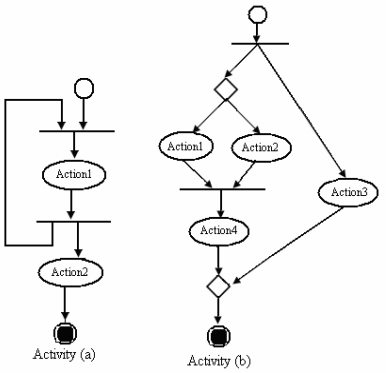

Fig. 8 Two faulty Activities (a) contains a deadlock (b) contains an unreachable *Action* node

Now, consider the Activity of Fig. 8 (b). If we replace the name of *Action* node in Fig. 7 (*ActionWithToken*) with "Action4", then check the property, it is not satisfied. Because there is a *Decision* node before "*Action1"* and "*Action2"*, according to the semantics of *Decision* nodes, the token is routed to only one of them. Hence in all execution of this Activity, only one token will arrive at *Join* node before *"Action4"*. Based on the semantics of *Join* nodes, token will never reach to the *"Action4"*. Therefore, we can conclude that this property is never satisfied for this workflow. We can state this kind of property in a more general way by ignoring the name of *Action*. In this case, checking the property only says that is there any unreachable *Action* in the Activity or not, and it does not determine the unreachable *Action* itself.

V. Conclusion and Future Work

In this paper we have presented an approach to formally define a semantics for UML 2.0 Activities. We have defined this semantics based on "token flow" and "traverse-to-completion" using graph transformation systems. To implement static semantics, we have defined a type graph (based on UML 2.0 Activity metamodel) and all Activities are modeled as a host graph. The host graph must confirm to the type graph. To define dynamic semantics, we have defined some graph transformation rules. We have used our previous

[6] In this case, a transformer is needed to automatically transform UML Activities to graph transformation

approach to verify graph transformations. As workflows are a typical modeling domain for UML 2.0 Activities, we have illustrated our proposed verification approach to verify workflows by defining some quality criterion. Non-expert designers can use our approach without any knowledge about underlying formalisms (i.e. BIR and Bogor).

However, further research is required to model other required elements (e.g. *Parameter* nodes and *Pins*). Our approach has the ability to support data flow, so we have a plan to model other elements and implement their semantics as some graph transformation rules.

It remains to discuss the size of Activities (or graphs in general) that our approach can verify, because in all model checking approaches state space explosion is a serious restriction. Scalability of our approach depends on different parameters: the size of the host graph, the number of dynamic nodes which must be added/deleted to/from the host graph by rules, the number of applicable rules in the same time, etc. We know there is not any dynamic node in the Activities. In addition, usually the size of Activities is not too large, even in the cases that the size of Activities is too large (e.g. Activities with more than 100 nodes) designers can decrease the size of them using "*Action Call*" nodes. Hence, we believe that our approach can support Activities with reasonable size.

ACKNOWLEDGMENT

This research was partially done while the first author was in university of Politecnico di Milano (Italy) as a visiting researcher and would like to thank the supports provided by Professor Luciano Baresi and Dr. Paola Spoletini.

REFERENCES

[1] Eshuis, R., Jansen, D. and Andwieringa, R.: Requirements-level Semantics and Model Checking of Object-Oriented Statecharts. Requirements Eng. J. 7, 243–263, (2002)
[2] Alonso, G., Casati, F., Kuno, H. and Machiraju, V.: Web Services: Concepts, Architectures and Applications. Springer, (2004)
[3] Object Management Group: UML Specification V2.0. http://www.omg.org/ technology/documents/modeling spec catalog.htm (2005)
[4] Baresi, L. and Heckel, R.: Tutorial Introduction to Graph Transformation: A Software Engineering Perspective, In proc. of first International Conference on Graph Transformation (ICGT), vol 2505 of LNCS, 402-429, (2002)
[5] Ehrig, H., Engels, G., Kreowski, H.j. and Rozenberg, G. (eds.): Handbook on Graph Grammars and Computing by Graph Transformation, vol. 2: Applications, Languages and Tools. World Scientific, (1999)
[6] Kuske, S.: A Formal Semantics of UML State Machines Based on Structured Graph Transformation, In Proc. of UML 2001, vol. 2185, Springer-Verlag, (2001)
[7] Beyer, M.: AGG1.0 – Tutorial. Technical University of Berlin, Department of Computer Science, (1992)
[8] Baresi, L., Rafe, V., Rahmani, A.T. and Spoletini, P.: An Efficient solution for Model Checking Graph Transformation Systems. Electronic Notes in Theoretical Computer Science (ENTCS), Vol. 213, PP. 3-21, (2008)
[9] Robby, Dwyer, M. and Hatcliff, J.: Bogor: An Extensible and Highly-Modular Software Model Checking Framework, In Proc. of the 9th European software engineering Conference, 267–276, (2003)
[10] Eshuis, R.: Semantics and Verification of UML Activity Diagrams for Workflow Modelling, Ph.D. Thesis, University of Twente, Netherlands, (2005)
[11] Bolton, C., Davies, J.: On Giving a Behavioural Semantics to Activity Graphs. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000. vol. 1939 of LNCS, Springer, Heidelberg (2000)
[12] Soltenborn, C.: Analysis of UML Workflow Diagrams with Dynamic MetaModeling Techniques, Master's Thesis, University of Paderborn, Germany,(2006)
[13] Hausmann, J. H.: Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages, Ph.D. Thesis, University of Paderborn, Germany, (2005)
[14] Engels, G., Soltenborn, C. and Wehrheim, H.: Analysis UML Activities Using Dynamic Meta Modeling, In Proc. of 9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS), vol 4468 of LNCS, 76-90, (2007)
[15] Rensink, A.: The GROOVE Simulator: A Tool for State Space Generation, In Applications of Graph Transformations with Industrial Relevance (AGTIVE), vol. 3062 of Lecture Notes in Computer Science, 479-485, (2004)
[16] Störrle, H., Hausmann, J.H.: Towards a Formal Semantics of UML 2.0 Activities. In: Liggesmeyer, P., Pohl, K., Goedicke, M. (eds) Software Engineering. LNI., GI, vol. 64 pp. 117–128 (2005)
[17] Eshuis, R.: Symbolic Model Checking of UML Activity Diagrams. ACM Transaction on Software Engineering Methodology, 15(1), 1–38 (2006)
[18] Cimatti, A., Clarke, E., Giunchiglia, F. and Roveri, M.: NuSMV: A New Symbolic Model Checker," International Journal on Software Tools for Technology Transfer, 2(4):410–425, (2000)
[19] Börger, E., Cavarra, A., Riccobene, E.: An ASM Semantics for UML Activity Diagrams. In: Rus, T. (ed.) AMAST 2000. vol. 1816 of LNCS, pp. 293–308. Springer, Heidelberg (2000)
[20] Baldan, P., Corradini, A., and Gadducci, F.: Specifying and Verifying UML Activity Diagrams via Graph Transformation. In Proc. of Global Computing, vol. 3267 of LNCS, 18-33, (2004)
[21] Störrle, H.: Semantics of Control-Flow in UML 2.0 Activities, In: N.N., editor, Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC) (2004)
[22] Bock, C.: UML 2 Activity and Action Models Part 4: Object Nodes, In Journal of Object Technology, vol. 3, no. 1, pp. 27-41. (2004)

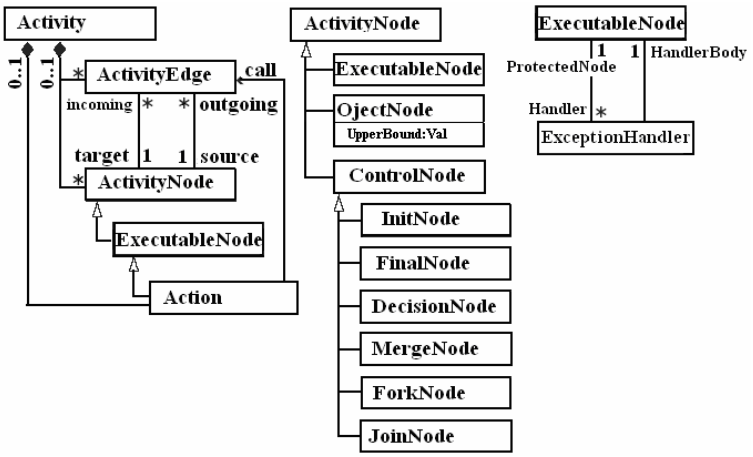

Fig. 1 A small portion of UML 2.0 metamodel [21]

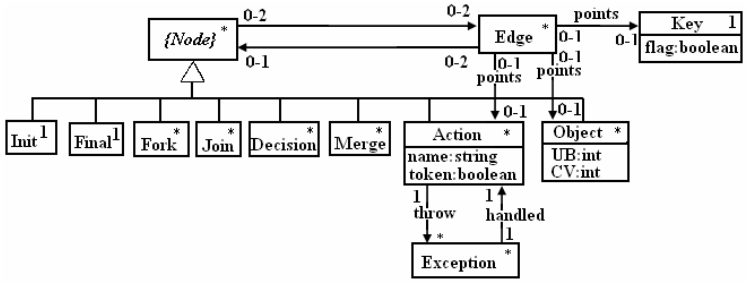

Fig. 2 Proposed type graph for UML 2.0 Activity diagram

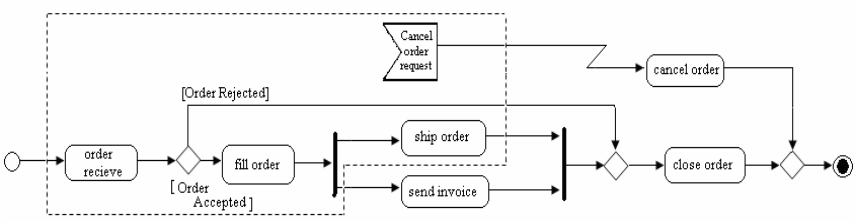

Fig. 4 A sample Activity diagram [3]