A Framework for Verifying UML Behavioral Models

Elena Planas*

Estudis d'Informàtica, Multimèdia i Telecomunicació
Universitat Oberta de Catalunya
eplanash@uoc.edu

Abstract. MDD and MDA approaches require capturing the behavior of UML models in sufficient detail and precision so that the models can be automatically implemented/executed in the production environment. With this purpose, Action Semantics were added to the UML specification as the fundamental unit of behavior specifications. Actions are the basis for defining the fine-grained behavior of operations, activity diagrams, interaction diagrams and state machines. Unfortunately, most of the current proposals devoted to the verification of behavioral models tend to skip the analysis of the actions they may include. The main goal of this PhD is to cover this gap by proposing a new verification framework aimed at verifying action-based behavioral specifications. In particular, we plan to describe several correctness properties of these specifications, develop a set of verification techniques based on the static analysis of the actions included in the action-based behavioral specifications for verifying these properties and integrate our techniques with other existing verification approaches.

1 Context

One of the most challenging and long-standing goals in software engineering [21] is to automate as much as possible the software development process. In fact, the software engineering community envisages a future in which, of all the phases of software development, software engineers will only be strictly necessary during the specification of the information system, while the remaining phases (mainly design, implementation and test) would be fully automated. This is also the focus of some of the most popular and current development methods as the MDD (Model-Driven Development [1]) and MDA (Model-Driven Architecture [18]) approaches.

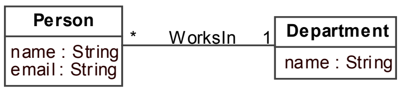
Models are a key artifact in development process and, thus, their correctness has a direct effect on the quality of the final system implementation. Wrong models can lead to incorrect implementations. Here is where the verification methods come into play. There are many methods for verifying models. The kind of verification depends on the type of the model and on the property we want to verify. Most of the current methods are focused on the static part of the models (for example, checking consistency

* Under the supervision of Dr. Jordi Cabot and Dra. Cristina Gómez.

between structural parts of the model) but less work has been done with respect to the dynamic part (part of the model that defines the system behavior), which is the topic of this thesis proposal.

2 Motivation

MDD, MDA and all executable UML methods [17] require capturing the behavior of UML models in sufficient detail and precision so that the models can be automatically implemented/executed in the production environment. In order to achieve this purpose, Action Semantics were added to the UML specification [20] as the fundamental unit of behavior specification. Actions are the basis for defining the fine-grained behavior of behavioral models (operations, activity diagrams, interaction diagrams and state machines). Actions enable to modify the system state through basic modifications (insert/update/delete) on the system elements (classes, associations and so on). Their resolution and expressive power are comparable to the executable instructions in traditional programming languages.



```
context Person::changeAddress(a:String) {
   AddStructuralFeature(self,address,a); }

context Person::addPerson(n:String, e:String) {
  p: Person;
  p := CreateObject(Person);
  AddStructuralFeature(p,name,n);
  AddStructuralFeature(p,email,e); }
```

Fig. 2.1. A simple example of a class diagram with two operations.

As a simple example, consider the class diagram (static part of a model) of Fig. 2.1 that describes the objects within a system (*people* and *departments*) and their relationships (person *works in* a department). The system behavior is defined by means of two operations specified with Action Semantics (*changeAddress* and *addPerson*). In this context, both operations are incorrect, since *changeAddress* tries to update an attribute (*address*) which does not even exist in the class diagram and *addPerson* can never be successfully executed (i.e. every time we try to execute *addPerson* the new system state violates the minimum '1' cardinality constraint of the *department* role in *WorksIn*, since the created person instance *p* is not linked to any department). Besides, this operation set is not *complete*, i.e. through these operations users cannot modify all elements of the class diagram, e.g. it is not possible to create and destroy departments. If these errors are not fixed before continuing with the code-generation phase, the resulting system implementation will be totally useless.

Most of the current verification approaches of behavioral models are mainly focused on state machines and they ignore the actions included in them, so we will focus on this gap.

3 Our Proposed Approach

The main goal of this PhD is to provide a verification framework to help the designers to verify the correctness of their behavioral models. The framework is mainly focused on the analysis of the modification actions included in the specification of those models and on how this analysis can complement other existing verification approaches.

3.1 Sub-Goals

The framework presented in this thesis proposal can be outline in the following sub-goals:

1. Identify and describe several fundamental correctness properties of action-based behavioral specifications: *syntactic correctness*, *executability*, *completeness* and *redundancy*.
2. Develop a static model verifier composed by a set of efficient techniques to verify the previous properties. Each of our techniques is focused on the verification of a particular correctness property. This set of techniques works at model level and performs a static analysis (i.e. our techniques do not require a simulation of the model) based on the study of the dependencies [3] among the actions included in the behavioral specification. These techniques must take into account the interactions between actions and the rest of the elements of the diagram where they are included (state machines, activity diagrams or interaction diagrams).
3. Integrate our static techniques with other existing verification approaches. Our static techniques may be complemented with existing verification approaches, for instance, those based on model checking. In particular, model checking techniques could be useful to get a more fine-grained analysis of the executability property once our first static analysis has determined that the specification is at least weak executable (i.e. there is at least a chance that it can be successfully executed). Depending on the verification technique that we apply, we have a different trade-off between the properties of the verification (efficiency, simplicity, result provided and so on). As part of our work, the framework has to be able to recommend the most suited technique/s depending on the given behavior specification, the property to verify and the constraints (time, precision,…) for that verification.
4. Provide useful feedback to the designer. The success of the application of formal techniques lies in providing the user with an understandable feedback (e.g. the feedback must be expressed in the same language used to model the behavior).

3.2 Overall framework architecture

Fig. 3.2.1 shows the overview of the framework architecture that we propose. First, several correctness properties can be checked by our static model verifier on the input behavior specification provided by the designer. In case of an error, the verifier returns a meaningful feedback that helps repairing the possible inconsistencies. After this first analysis, some of these properties may be also deal with current verifying techniques as model checking. Particularly, our static techniques could be used to perform a first correctness analysis, basic to ensure a fundamental quality level in action specifications. Then, designers could proceed with a more detailed analysis adapting current approaches as model checking. For instance, example execution traces that lead to an error state provided by model checking techniques would help designers to detect particular scenarios not yet appropriately considered.

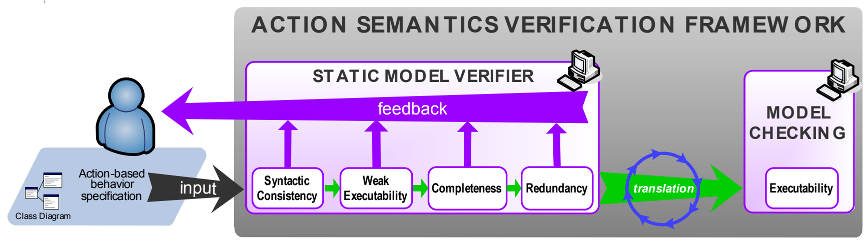


Fig. 3.2.1. Action Semantics verification framework overview.

Next, we describe on more detail each of the components of the framework.

Input

The input of our verification techniques is an action-based behavioral specification that may include several UML actions and the corresponding Class Diagram. The main modification actions that may appear in the action specification are:

1. *CreateObject*: Creates a new object that conforms to the specified classifier. The object is returned as an output parameter.
2. *DestroyObject*: Destroys an object.
3. *AddStructuralFeature*: Sets a value for an attribute of an object.
4. *CreateLink*: Creates a new link in a binary association.
5. *DestroyLink*: Destroys the link between two objects from an association.
6. *ReclassifyObject*: Adds an object as a new instance of a set of new classes and removes it from old classes.
7. *CallOperation*: Invokes an operation on an object with arguments and returns the results of the invocation.

These actions can be accompanied with several read actions (e.g. to access the values of attributes and links of the objects) and can be structured to coordinate basic actions in action sequences, conditional blocks or loops (*do-while* or *while-do* loops) to completely define the effect of behavioral specifications.

Correctness properties

The correctness properties that we deal with can be summarized in:

- ***Syntactic correctness***: Concerning UML models, syntactical correctness ensures that a specification conforms to the abstract syntax specified by the UML metamodel [20] (similar to the grammar of programming languages). For instance, suppose an abstract class *A* and the action *obj:=CreateObject(A)*. This action is not syntactically correct due to we cannot create an instance of an abstract class. Syntactical consistency conditions are expressed in UML metamodel using a set of (OCL) constraints (i.e. Well-Formedness Rules, WFR) that restrict the possible set of valid (i.e. Well-Formed) UML models. These WFRs help to prevent syntactic errors in action specifications. We say that an action-based behavior specification is *syntactically correct* when all actions included in it conform to these WFR.
- ***Executability***: We consider that an action-based behavior specification is *weakly executable* when there is a chance that a user may successfully execute the behavior, that is, when there is at least an initial system state for which the execution of the actions included in the behavior evolves this state to a new system state that satisfies all integrity constraints of the model. Otherwise, the behavior is completely useless. We define our executability property as *weak executability* since we do not require all executions of the behavior to be successful, which could be defined as *strong executability*. For instance, in the operations context, the operation *addPerson* of Fig. 2.1 is *not-weakly-executable* since, every time we try to execute it, the new system state violates the minimum '1' cardinality constraint of the *department* role in *WorksIn*, because the created person instance *p* is not linked to any department.
- ***Completeness***: We consider that a set of action-based behavior specifications is *complete* when all possible changes on the system state can be performed through the execution of this set of behaviors. Otherwise, there will be parts of the system that users will not be able to modify since no available behavior address their modification. For instance, in the operations context, the operation set of Fig. 2.1 is *not complete*, i.e. through these operations users cannot modify all elements of the class diagram, e.g. it is not possible to create/destroy *departments*.
- ***Redundancy***: An action (or set of actions) in an action-based behavior specification is *redundant* if its effect on the system state is subsumed by the effect of later actions in the same action-based behavior specification, that is, the final system state when executing the behavior would be exactly the same with or without the redundant action.

Feedback

Our static verification techniques should provide a valuable feedback since, for each detected error, they suggest to the designer how to change the specification in order to repair the detected inconsistency. For instance, if check the executability of the operation *addPerson* (Fig. 2.1), our techniques should report to the designer that adding a new link to the dangling object (with *CreateLink(WorksIn,...)*) or destroying it (with *DestroyObject(self)*) would make the operation executable.

Model Checking

As we have seen, after a first static analysis, some correctness properties may be also deal with current verifying techniques as model checking, which performs a more detailed (but less useful in terms of corrective feedback provided) analysis of each executable trace.

A model checking analysis of an action-based behavioral specification requires translating this specification into an input language that can be understood by current model checking techniques (for instance, Maude [5] or SPIN [13]).

4 Research Plan

In order to achieve our objectives, we plan structuring our research in the following main stages:

1. *Correctness properties definition*: In this stage, we plan to define the correctness properties that we plan to verify.
2. *Definition of techniques for static verification of previous properties*: This stage comprises the elaboration of static techniques for verifying each property. Firstly, our techniques will be centered on action sequences part of individual operations and, next, we plan to extrapolate and integrate the techniques for verifying more complex action sequences that are part of complex diagrams (activity diagrams, interaction diagrams and state machines).
3. *Development of a verification tool*: The next stage is to implement a tool that supports completely our static techniques.
4. *Integration of our techniques with current verification methods*: We plan to integrate our techniques with other existing verification methods for behavior specifications. Note that this stage includes performing the transformation from a UML action-based sequence to the input languages of other methods.
5. *Techniques analysis*: Once we have our static techniques and current model checking techniques, we plan to compare both technique types, analyze its trade offs (precision, efficiency, and so on) and recommend when to use the model checking techniques (or other verification techniques) depending on the model complexity and the result of the first analysis.
6. *Framework validation*: Finally, we plan to validate our framework in real case scenarios. In particular, we plan to test the usefulness of each technique, the feedback comprehension and the results obtained using our techniques.

So far, we have addressed step 1, and part of step 2 considering just action sequences as input [22].

5 Related work

There is a wide set of research proposals devoted to the verification of behavior specifications in UML, mainly focusing on state machines [15], [16], [19], interaction diagrams [2], sequence diagrams [12], activity diagrams [10] or on the consistent interrelationship [14], [7], [11], [26], [25], [23], [8] and no-redundancies [6] between them and/or the class diagram, among others. Nevertheless, many of these methods target very basic correctness properties (basically some kind of well-formedness rules between the different diagrams) and/or restrict the expressivity of the supported UML models. Moreover, most of the methods above ignore the semantics of the actions included in the specifications (a relevant exception is [24]), which is exactly the focus of our approach.

Besides, to check the executability of a behavior specification (or, in general, any property that can be expressed as a Linear Temporal Logic formula - LTL [9]) previous approaches rely on the use of model checking techniques [13]. Roughly, model checkers work by generating and analyzing all the potential executions at run-time and evaluating if, for each (or some) execution, the given property is satisfied. Model checking techniques, unlike the static techniques, suffer from the state-explosion problem [4] (i.e. the number of potential executions to analyze grows exponentially in terms of the size of the model, the domains of the parameters,…) even though a number of optimizations are available (as partial order reduction or state compression).

A significant difference between model checking techniques and the static techniques that we propose is its output. model checking based proposals provide example execution traces that do (not) satisfy the checked property. On the other side, our static techniques suggest how to change the operation specification in order to repair the detected inconsistency. The static techniques that we propose provide a more valuable feedback for a first correctness analysis, but model checking techniques could also be used to get more information (e.g. incorrect execution traces). For this reason, both technique types (model checking and static techniques) may be integrated in a framework for performing a complete verification.

We would like to remark that, to the best of our knowledge, our approach is the first one considering the completeness, redundancy and syntactic analysis of action specifications.

Acknowledgements. Thanks to the anonymous referees for their useful comments to previous drafts of this paper. This work has been partly supported by the Ministerio de Ciencia y Tecnologia under TIN2008-00444 project, Grupo Consolidado.

References

1. Atkinson, C., Kühne, T.: Model-Driven Development: A Metamodeling Foundation. IEEE Software. (2003)

2. Baker, P., Bristow, P., Jervis, C., King, D., Thomson, R., Mitchell, B., Burton, S.: Detecting and Resolving Semantic Pathologies in UML Sequence Diagrams. ESEC/SIGSOFT FSE, 50-59 (2005)
3. Cabot, J., Gómez, C.: Deriving Operation Contracts from UML Class Diagrams. MoDELS, LNCS, 4735, 196-210, (2007)
4. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Progress on the State Explosion Problem in Model Checking. Informatics-10 Years Back, 10, 176-194 (2001)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Messeguer, J., Talcott, C.: Maude Manual (Version 2.4) (2009)
6. Costal, D., Sancho, M. R., Teniente, E.: Understanding Redundancy in UML Models for Object-Oriented Analysis. CAiSE, 659-674 (2002)
7. Gallardo, M.M., Merino, P., Pimentel, E.: Debugging UML Designs with Model Checking. Journal of Object Technology, 1(2), 101-117 (2002)
8. Egyed, A.: Instant Consistency Checking for the UML. ICSE, 381-390 (2006)
9. Emerson, E. A.: Temporal and Modal Logic. Handbook of Theoretical Computer Science, 8, 995-1072 (1990)
10. Eshuis, R.: Symbolic Model Checking of UML Activity Diagrams. ACM Transactions on Soft. Eng. and Methodology, 15, 1-38 (2006)
11. Graw, G., Herrmann, P.: Transformation and Verification of Executable UML Models. Electronic Notes in Theoretical Computer Science, 101, 3-24 (2004)
12. Grosu, R., Smolka, S. A.: Safety-Liveness Semantics for UML 2.0 Sequence Diagrams. ACSD, 6-14 (2005)
13. Holzmann, G. J.: The spin model checker: Primer and reference manual. Addison-Wesley Professional (2004)
14. Knapp, A., Wuttke, J.: Model Checking of UML 2.0 Interactions. MoDELS Workshops, LNCS, 4364, 42-51 (2006)
15. Latella, D., Majzik, I., Massink, M.: Automatic Verification of a Behavioural Subset of UML Statechart Diagrams using the SPIN Model-Checker. Formal Aspects of Computing, 11(6), 637-664 (1999)
16. Lilius, J., Paltor, I. P.: Formalising UML State Machines for Model Checking.UML, LNCS, 1723, 430–445 (1999)
17. Mellor Stephen J., Balcer Marc J.: Executable UML: A foundation for model-driven architecture. Addison-Wesley (2002)
18. Mellor, S. J., Scott, K., Uhl, A., Weise, D.: Model-Driven Architecture. Computing Reviews, 45, 631 (2004)
19. Ober, I., Graf, S., Ober, I.: Validating Timed UML Models by Simulation and Verification. Int. Journal on Software Tools for Technology Transfer, 8(2), 128-145 (2006)
20. Object Management Group (OMG): UML 2.0 Superstructure Specification. OMG Adopted Specification (ptc/07-11-02) (2007)
21. Olivé, A.: Conceptual Schema-Centric Development: A Grand Challenge for Information Systems Research.CAiSE, 3520 (2005)
22. Planas, E., Cabot, J., Gómez, C.: Verifying Action Semantics Specifications in UML Behavioral Models. CAiSE (2009). To appear.
23. Rasch, H., Wehrheim, H.: Checking Consistency in UML Diagrams: Classes and State Machines. FMOODS, LNCS, 2884, 229-243 (2003)
24. Turner E., Treharne H., Schneider S., Evans N.: Automatic Generation of CSP || B Skeletons from xUML Models. ICTAC, LNCS, 364-379 (2008)
25. Van Der Straeten, R., Mens, T., Simmonds, J., Jonckers, V.: Using Description Logic to Maintain Consistency between UML Models. UML, LNCS, 2863, 326–340 (2003)
26. Xie, F., Levin, V., Browne, J. C.: Model Checking for an Executable Subset of UML. ASE, 333-336 (2001)