Lightweight Verification of Executable Models

Elena Planas[1], Jordi Cabot[2], and Cristina Gómez[3]

[1] Universitat Oberta de Catalunya (Spain), eplanash@uoc.edu
[2] École des Mines de Nantes - INRIA (France), jordi.cabot@inria.fr
[3] Universitat Politècnica de Catalunya (Spain), cristina@essi.upc.edu

Abstract. Executable models play a key role in many development methods by facilitating the immediate simulation/implementation of the software system under development. This is possible because executable models include a fine-grained specification of the system behaviour. Unfortunately, a quick and easy way to check the correctness of behavioural specifications is still missing, which compromises their quality (and in turn the quality of the system generated from them). In this paper, a lightweight verification method to assess the strong executability of fine-grained behavioural specifications (i.e. operations) at design-time is provided. This method suffices to check that the execution of the operations is consistent with the integrity constraints defined in the structural model and returns a meaningful feedback that helps correcting them otherwise.

1 Introduction

Executable models play a cornerstone role in the Model-Driven Development (MDD) approach, in which models must be fine-grained specified in order to be used to (semi)automatically implement the software system. Executable models are now increasing its popularity. As a relevant example, the OMG has recently published the first version of the fUML [15] standard, an executable subset of the UML [13] that can be used to define, in an operational style, the structural and behavioural semantics of systems.

In MDD the quality of the final system implementation depends on the quality of the initial specification, so the existence of methods to verify the correctness of executable models is becoming crucial. In this sense, the goal of this paper is to propose a lightweight verification method for executable models. Our method focuses on the verification of the *strong executability* correctness property of action-based operations. An operation is *Strongly Executable* (SE) if it is guaranteed that every time the operation is invoked, the set of modifications the operation performs on the system´s data evolves the system to a new state fully consistent with all integrity constraints. This is one of the most fundamental correctness properties for behavioural specifications. When we know that all operations are SE, we can avoid checking at the end of each operation execution if all constraints are satisfied which improves the efficiency of the system.

In this paper we assume structural models are written in UML/OCL and operations are specified in Alf Action Language [14], although our method could be used with models written in other languages.

Paper Organization. The rest of the paper is structured as follows. Section 2 presents the state of the art. Section 3 introduces several preliminary concepts. Section 4 presents our method and describes the feedback it provides. Finally, Section 5 presents our conclusions and further work.

2 State of the Art

There is a broad set of research proposals devoted to the verification of (UML) behavioural models, focusing on state machines [11] [12], sequence diagrams [3] [9], activity diagrams [1] [6], operations [4] [18] [16], or the consistent interrelationship between them and/or the class diagram [8], among others.

Table 1 classifies the most representative works and positions our method wrt them. For each approach we indicate the kind of behavioural model targeted, the integrity constraints that are considered when evaluating the models, whether actions can be added to specify the model, the main correctness properties addressed by the method, the basic technique employed during the verification and if the approach returns some kind of feedback beyond a simple yes/no answer.

Table 1. Related methods comparison. *Abbreviations: AD(Activity Diagram), SqD(Sequence Diagram), StD(Statechart Diagram), decl-OP(Declarative Operation), imp-OP (Imperative Operation), MC(Model Checking), CP(Constraint Programming), QC (Query Containment).*

Refs	*Behavioural Model*	*Supported Constraints*	*Include Actions?*	*Main Properties*	*Technique*	*Repairing Feedback?*
[8]	AD,SqD,StD	None	Yes	Consistency	MC	No
[1]	AD	None	Yes	Deadlocks	MC	No
[11]	StD	None	No	Safety Liveness	MC	No
[12]	StD	Associated to states	No	Deadlocks Livelocks	MC	No
[4]	decl-OP	All	No	SE et al.	CP	No
[18]	decl-OP	Subset	No	Weak Executability et al.	QC	No
our work	imp-OP	Subset	Yes	SE	Static Analysis	Yes

Only few works [8] [1] (and also our previous work [16]) allow inclusion of actions in their diagrams, which is precisely the focus of our method, but they do not check their strong executatibility. Works dealing with the executability of operations [4] [18] depart from declarative operations specified by means of pre and postconditions contracts, instead of using imperative specifications.

The above works simulate the behavioural models by translating them into Model Checking [2], Constraint Programming [10] or Query Containment [7], and thus, do not scale properly and compromise the efficiency of the method. Instead, our method performs an static analysis, thus, it does not require to simulate the behaviour in order to determine their executability.

On the other hand, most of the above methods do not provide a valuable feedback but just provide a binary response and, at most some provide example execution traces that do (not) satisfy the property. None clearly identify the source of the problems nor assist the designer to repair them.

As a trade-off, our method supports a restricted set of integrity constraints. Once the model passes our first analysis, designers may choose to use more expressive (but more time expensive) methods.

3 Preliminary Concepts

This section introduces some preliminary concepts to facilitate the comprehension of our method and presents a running example to illustrate these concepts.

3.1 Structural Model

Class Diagram. A UML Class Diagram (CD) consists of a set of classes (i.e. entity types), attributes, associations (i.e. relationship types) among classes, generalizations among classes and integrity constraints.

Integrity Constraints. When verifying executability of operations, our method covers the constraints that most frequently appear in the structural models (a detailed list of the addressed constraints is provided in [17]).

Example. Figure 1 shows an excerpt of the class diagram and integrity constraints in an e-commerce system. It includes information about its *customers*, which can acquire *orders*. An *order* is composed of one or more *order lines*, each of them related to exactly one *product*. A product may be in offer as denoted by the value of the *special price* attribute. A product may have several substituted products, which are suggested to the customer when the desired product is sold out. Additional identifier, value comparison and symmetric constraints are expressed in OCL. The derivation rule for *totalPrice* attribute is not showed here since it has no relevance in our analysis.

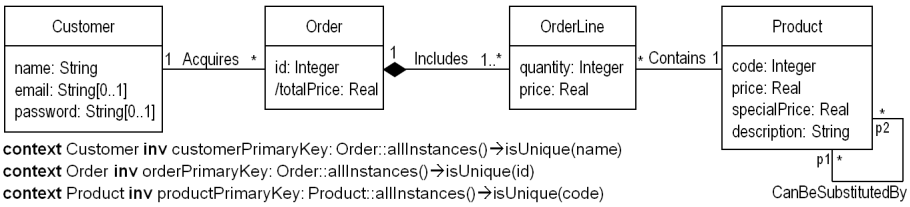

Fig. 1. Excerpt of a e-commerce system class diagram.

3.2 Behavioural Model

In UML there are several alternatives to represent the behaviour of a system but the basic way is using operations (attached into classes) that the user may execute to query and/or modify the information modeled in the structural model.

Our method is focused on operations specified by means of *"Action Language for fUML"* (Alf) [14], a beta standard published recently by the OMG. Alf

provides a concrete syntax conforming to the fUML abstract syntax, defining the basic actions to specify the fine-grained behaviour of systems and a set of statements to coordinate these actions in action sequences, conditional blocks or loops. As any action language, the expressiveness of Alf is comparable to that of the instructions in traditional programming languages but at a higher abstraction (and platform-independent) level.

Example. We show an operation[1] of the e-commerce system defined using Alf language: "newProduct", which creates a new product in the system.

```
activity newProduct(in _code:String, in _price:Real, in _specialPrice:Real, in _description:String,
in _substitutedProducts:Product[0..*]){
  p = new Product();
  p.code = _code;
  p.price = _price;
  p.specialPrice = _specialPrice;
  p.description = _description;
  for (i in 1.._substitutedProducts→size()) {
   CanBeSubstitutedBy.createLink(p1=>self,p2=>_substitutedProducts[i]);
  }
}
```

4 Our Method

Our method aims to verify at design-time if an action-based operation is *Strongly Executable* (SE). We consider that an operation is SE if it is always successfully executed, i.e. if every time we execute the operation (whatever values are given as arguments for its parameters), the effect of the actions included in the operation evolves the initial state of the system to a new system state that satisfies all integrity constraints of the structural model.

Given an input structural and behavioural models, our method (see Figure 2) returns either a positive answer (meaning that the operation is SE) or a corrective feedback. This corrective feedback consists in a set of actions and conditions that should be added to the operation in order to make it SE. Extending the operation with this feedback is a necessary condition but not a sufficient one to immediately guarantee the SE of the operation since the added actions may in its turn induce other constraint violations. Therefore, the extended operation must be recursively reanalyzed with our method until we reach a SE status.

When analyzing the SE of the operation, we must take into account all possible *execution paths* (an operation is SE iff all its paths are SE). Therefore, the first step of the method is to compute such paths (Section 4.1). Once the paths have been computed, the rest of the method is applied on each path. Step 2 (Section 4.2) analyzes individually each action in the path p to see if it may violate some integrity constraint of the structural model. Finally, Step 3 (Section 4.3) performs a contextual analysis of each potentially violating action to see if other actions or conditions in p compensate or complement its effect to ensure that we always reach a consistent state at the end of the operation. If all potential violation actions can be discarded we can conclude that p is SE.

[1] Operation methods are specified as UML activities in Alf.

Our method performs an over-approximation analysis. This implies that it may return *false positives*, that is, it may return as a non-SE an operation which is actually SE. The designer intervention is necessary to confirm non-SE in those cases. On the other hand, the method does not return *false negatives* (in our opinion, more critical than the above), that is, when it states that an operation is SE, this statement is always true. Over-approximation is due to the lack of exhaustiveness in the comparison of conditions in the operation to favour the efficiency of the process. We believe this is a reasonable trade-off for the method.

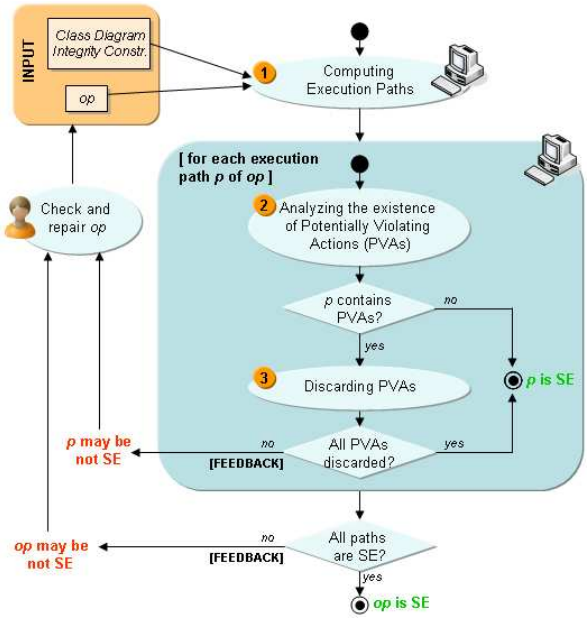

Fig. 2. Method overview.

4.1 Step 1: Computing the Execution Paths of the operation

An *execution path* of an operation *op* is a consecutive sequence of actions that may be followed during the execution of *op* in a given execution scenario. For trivial operations (e.g. with neither conditional or loop structures) there is a single execution path but, in general, several ones will exist.

We propose to represent each operation as a directed graph. Then, execution paths are all paths in the graph that start at the initial vertex, end at the final vertex and does not include repeated arcs.

Example. Figure 3 shows the directed graph of operation "newProduct". Two execution paths may be derived: the first one, executed when the new product has no substituted products (the loop is not executed); and the second one, executed otherwise.

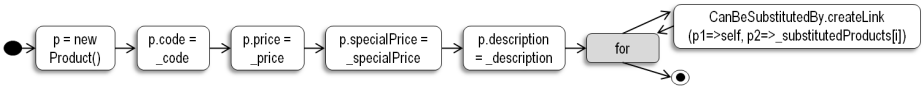

Fig. 3. Execution paths of "newProduct".

4.2 Step 2: Analyzing the existence of Potentially Violating Actions

This step analyzes each action in the path to see if its effect can change the system state in a way that some integrity constraint becomes violated. If so, this action is declared as a *Potentially Violating Action* (PVA) and we refer to the constraints the PVA can violate as *Susceptible Violated Constraints* (SVC). If the path has no PVAs, is SE. Otherwise, we need to continue the analysis with the next step.

To detect PVAs we rely on the method published in [5] that receives as input a CD and a set of constraints and automatically determines the actions that may violate each constraint of the model. Thus, we may determine if a path contains PVAs by comparing the list of actions in the path with the list of actions returned by this method. All actions in the intersection of both sets are PVAs.

Example. Second path of "newProduct" (first path is not explicitly shown since it is a subset of this one) contains four PVAs: PVA_1, which may violate four mandatory constraints (when the attributes *code*, *price*, *specialPrice* and *description* are not initialized); PVA_2, which may violate the *productPrimaryKey* constraint (when the system contains another product with the same *code*); PVA_3, which may violate the *specialPrice* constraint (when *_specialPrice* $\geq self.price$); and PVA_4 which may violate the *symmetricAssociation* constraint (when the opposite link is not created).

Potentially Violating Actions of "newProduct" (second path):
- PVA_1: p = new Product()
  - $SVC_{1.1}$: The attribute "code" of class "Product" must have at least one value
  - $SVC_{1.2}$: The attribute "price" of class "Product" must have at least one value
  - $SVC_{1.3}$: The attribute "specialPrice" of class "Product" must have at least one value
  - $SVC_{1.4}$: The attribute "description" of class "Product" must have at least one value
- PVA_2: p.code = _code
  - $SVC_{2.1}$: *productPrimaryKey* constraint
- PVA_3: p.specialPrice = _specialPrice;
  - $SVC_{3.1}$: *specialPrice* constraint
- PVA_4: CanBeSubstitutedBy.createLink(p_1=>self,p_2=>_substitutedProducts[i])
  - $SVC_{4.1}$: *symmetricAssociation* constraint

Since all paths of "newProduct" are susceptible to be non-SE (given that all of them contain some PVAs) we must proceed with the last step of our method.

4.3 Step 3: Discarding Potentially Violating Actions

It may happen that the context in which a PVA is executed within the path guarantees that the effect of the PVA is never going to actually violate any of its SVCs. Roughly, there are two ways to discard a PVA: (1) when the path contains a guard (i.e. a conditional structure) that ensures the PVA will only be executed in a safe context; and (2) when the path contains another action which counters or complements the effect of the PVA in order to maintain the integrity of the system. In this last step, we analyze the set of PVAs returned by the previous step and try to discard them by analyzing the two possibilities commented above. If all PVAs can be discarded, the path is classified as SE. If not, the path (and consequently the operation) is marked as non-SE and the corresponding corrective feedback is provided.

We have determined a set of *discarding conditions*, that is, the conditions that a path must satisfy in order to discard a specific PVA. Due to space limitations,

we do not show here these conditions (the interested reader may found it in [17]), but we illustrate a subset of these conditions using our running operation.

Example. In the following we try to discard the PVAs of our running path according to our discarding conditions. For each PVA and SVC, we show the conditions that the path should satisfy to discard that PVA. Then, $\{sat\}$ states that the condition is satisfied by the path, while $\{not\ sat\}$ states the opposite.

Conditions to discard the PVAs of "newProduct" (second path):
- PVA_1,$SVC_{1.1}$: The attribute "code" of object "p" must be initialized $\{sat\}$
- PVA_1,$SVC_{1.2}$: The attribute "price" of object "p" must be initialized $\{sat\}$
- PVA_1,$SVC_{1.3}$: The attribute "specialPrice" of object "p" must be initialized $\{sat\}$
- PVA_1,$SVC_{1.4}$: The attribute "description" of object "p" must be initialized $\{sat\}$
- PVA_2,$SVC_{2.1}$:
  - option 1: Must exist a guard which ensures the PVA will only be executed when there is not another product with the same "code" $\{not\ sat\}$
  - option 2: Must exist a product with the same "code" and its value is modified $\{not\ sat\}$
  - option 3: Must exist a product with the same "code" and this product is destroyed $\{not\ sat\}$
- PVA_3,$SVC_{3.1}$: Must exist a guard which ensures the PVA will only be executed when "_specialPrice<_price" $\{not\ sat\}$
- PVA_4,$SVC_{4.1}$: Must exist a creation of the symmetric link of type "CanBeSubstitutedBy" between objects "_substitutedProducts[i]" and "self" $\{not\ sat\}$

Second path of "newProduct" does not satisfy all discarding conditions, hence, our method concludes it is not SE (and the same for the first path).

In the following we show the repaired operation *newProduct* once the feedback provided by our method has been integrated. The added sentences are preceded by a right arrow (→). Each of them fixes one of the problems detected above. For the sake of simplicity, we only show one possible reparation.

```
activity newProduct(in _code:String, in _price:Real, in _specialPrice:Real, in _description:String,
in _substitutedProducts:Product[0..*] sequence){
→ if (not Product::allInstances()→exists(p|p.code=_code)) {
  → if (_specialPrice<_price) {
       p = new Product();
       p.code = _code;
       p.price = _price;
       p.specialPrice = _specialPrice;
       p.description = _description;
       for (i in 1.._substitutedProducts→size()) {
          CanBeSubstitutedBy.createLink(p1=>self,p2=>_substitutedProducts[i]);
       → CanBeSubstitutedBy.createLink(p1=>_substitutedProducts[i],p2=>self);
       }
  → }
→ }
 }
```

5 Conclusions and Further Work

We have proposed a lightweight method for assisting the designer during the specification of executable behavioural models. In particular, our method verifies the Strong Executability (SE) of action-based UML operations (although our method could be used with models written in other languages) wrt the integrity constraints imposed by the structural model at design-time. The main characteristics of our method are its efficiency (since no simulation/animation of the behaviour is required) and feedback (for non-executable operations, it is able to identify the source of the inconsistency and suggest possible corrections). For these reasons, our method is easy to integrate in existing CASE tools.

As a further work, we plan to study the executability of operations when they are combined with other UML behavioural diagrams and explore the integration of our verification method in a more complete verification framework that could help designers choose the most appropriate verification technique for the model they have defined, depending on the target property and the verification trade-offs (completeness, efficiency,...) they are ready to accept.

Acknowledgements. This work has been partly supported by the Ministerio de Ciencia y Tecnología under TIN2008-00444 project, Grupo Consolidado.

References

1. I. Abdelhalim, J. Sharp, S. A. Schneider, and H. Treharne. Formal Verification of Tokeneer Behaviours Modelled in fUML using CSP. In *ICFEM*, volume 6447 of *LNCS*, pages 371–387, 2010.
2. R. Alur. Model Checking: From Tools to Theory. In *25 Years of Model Checking*, pages 89–106, 2008.
3. P. Baker, P. Bristow, C. Jervis, D. J. King, R. Thomson, B. Mitchell, and S. Burton. Detecting and Resolving Semantic Pathologies in UML Sequence Diagrams. In *ESEC/SIGSOFT FSE*, pages 50–59, 2005.
4. J. Cabot, R. Clarisó, and D. Riera. Verifying UML/OCL Operation Contracts. In *IFM*, volume 5423 of *LNCS*, pages 40–55, 2009.
5. J. Cabot and E. Teniente. Determining the Structural Events That May Violate an Integrity Constraint. In *UML*, volume 3273 of *LNCS*, pages 320–334, 2004.
6. R. Eshuis. Symbolic Model Checking of UML Activity Diagrams. *ACM Transactions on Software Engineering and Methodology*, 15(1):1–38, 2006.
7. C. Farré, E. Teniente, and T. Urpí. Checking query containment with the CQC method. *Data Knowledge Engineering*, 53(2):163–223, 2005.
8. G. Graw and P. Herrmann. Transformation and Verification of Executable UML Models. *Electr. Notes Theor. Comput. Sci.*, 101:3–24, 2004.
9. R. Grosu and S. A. Smolka. Safety-Liveness Semantics for UML 2.0 Sequence Diagrams. In *ACSD*, pages 6–14. IEEE Press, 2005.
10. M. Hanus. Programming with Constraints: An Introduction by Kim Marriott and Peter J. Stuckey, MIT Press, 1998. *J. Funct. Program.*, 11(2):253–262, 2001.
11. D. Latella, I. Majzik, and M. Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. *Formal Asp. Comput.*, 11(6):637–664, 1999.
12. J. Lilius and I. Paltor. vUML: A Tool for Verifying UML Models. In *ASE*, pages 255–258, 1999.
13. OMG. UML 2.0 Superstructure Specification. (ptc/07-11-02), 2007.
14. OMG. Concrete Syntax for UML Action Language (Action Language for Foundational UML), version Beta 1, www.omg.org/spec/ALF, 2010.
15. OMG. Semantics Of A Foundational Subset For Executable UML Models (fUML), version 1.0, www.omg.org/spec/FUML, 2011.
16. E. Planas, J. Cabot, and C. Gómez. Verifying Action Semantics Specifications in UML Behavioral Models. In *CAiSE*, volume 5565 of *LNCS*, pages 125–140, 2009.
17. E. Planas, J. Cabot, and C. Gómez. Lightweight Verification of Executable Models (Extended Version), http://gres.uoc.edu/pubs/VerifyingExecModels.pdf, 2011.
18. A. Queralt and E. Teniente. Reasoning on UML Conceptual Schemas with Operations. In *CAiSE*, volume 5565 of *LNCS*, pages 47–62, 2009.