Lightweight and Static Verification of UML Executable Models

Elena Planas[a,*], Jordi Cabot[b], Cristina Gómez[c]

[a] *Universitat Oberta de Catalunya (Spain)*
[b] *ICREA at Internet Interdisciplinary Institute (IN3 - UOC) (Spain)*
[c] *Universitat Politècnica de Catalunya (Spain)*

Abstract

Executable models play a key role in many software development methods by facilitating the (semi)automatic implementation/execution of the software system under development. This is possible because executable models promote a complete and fine-grained specification of the system behaviour. In this context, where models are the basis of the whole development process, the quality of the models has a high impact on the final quality of software systems derived from them. Therefore, the existence of methods to verify the correctness of executable models is crucial. Otherwise, the quality of the executable models (and in turn the quality of the final system generated from them) will be compromised. In this paper a lightweight and static verification method to assess the correctness of executable models is proposed. This method allows to check whether the operations defined as part of the behavioural model are able to be executed without breaking the integrity of the structural model and returns a meaningful feedback that helps repairing the detected inconsistencies.

Keywords: Model-Driven Development (MDD), Model-Driven Architecture (MDA), Executable Models, Verification, Static Analysis, Alf Action Language

1. Introduction

Executable models are models with a behavioural specification detailed enough so that they can be systematically implemented or executed in the production environment. Executable models play a cornerstone role in the Model-Driven Development (MDD) paradigm, where models are the core artifacts of the development life-cycle and the basis to generate the final software implementation.

Executable models are not a new concept (e.g. [27, 51]) but are now experiencing a comeback, becoming a relevant topic within the OMG (Object

[*] *Corresponding author at:* Rambla del Poblenou 156, 08018 Barcelona, Spain. Tel.: +34 93 326 35 49; fax: +34 93 326 88 22.

Email addresses: eplanash@uoc.edu (Elena Planas), jordi.cabot@icrea.cat (Jordi Cabot), cristina@essi.upc.edu (Cristina Gómez)

Management Group). They use is being promoted given the value they can bring. As one of its creators declares *"executable models increase productivity by raising the level of abstraction; reduce costs by describing systems independently of their implementation; and improve the quality of the final system by facilitating early verification"* [32].

Following this trend, the OMG has published in the last years several versions of the *Foundational Subset for Executable UML Models* (fUML) standard [41], an executable subset of the UML that can be used to define, in an operational style, the semantics of systems. The OMG has also published the first standard version of the *Action Language for fUML* (Alf) standard [40], a concrete syntax conforming to the fUML abstract syntax, that provides the constructs and textual notation to specify the fine-grained behaviour of systems in terms of actions. The OMG support to executable models is also substantially raising the interest of software companies for this topic [17].

Given the increasing importance of executable models and the impact of their correctness on the final quality of software systems derived from them [35], the existence of methods to verify the correctness of such models is becoming crucial. Otherwise, the quality of the executable models (and in turn the quality of the final system generated from them) will be compromised.

Unfortunately, despite the number of research works targetting the verification of software models, their computational cost and poor feedback makes them difficult to be integrated in current software development processes. The goal of this paper is to propose a verification method that can overcome the limitations of the existing methods. Our approach (see Figure 1) focuses on the verification of the *executability* of operations specified by means of actions (i.e. action-based operations) with respect to a subset of the integrity constraints that can appear in a model. We consider this is one of the most fundamental properties of executable models since it guarantees the correctness of such models. Besides checking the executability of the operations, the method we propose returns a meaningful feedback that helps repairing the detected inconsistencies.

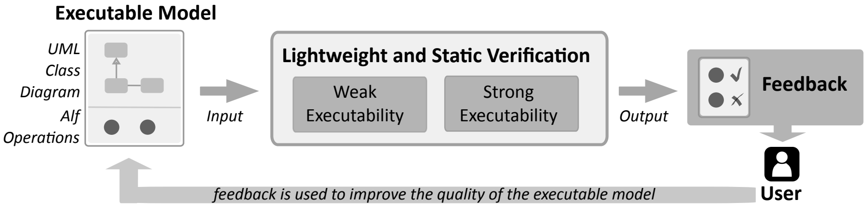


Figure 1: Method overview.

Like other studies that focus on executability [11, 12, 49], our method classifies the operations in three categories:

1. **Strongly executable (SE) operations**, i.e. operations that are guaranteed to *always* generate a consistent state. We know for sure that all executions of the operation (regardless of the input values provided to the operation and the initial system state where the operation is applied over)

reach a consistent state with respect to the structural model and their integrity constraints.

2. **Weakly executable (WE) operations**, i.e. operations that *sometimes* generate a consistent state, but are not guaranteed to do so. We can ensure that at least one of the many possible executions of the operation during the life span of the system will be successfully executed but probably not all of them (e.g. depending on the input parameters).
3. **Non executable (¬E) operations**, i.e. operations that *never* generate a consistent system state. After their execution, they always reach a state that violates some integrity constraints of the structural model (e.g. some cardinality constraints).

Clearly, ¬E operations are completely useless since every time a user tries to execute them (regardless the provided input values) an error arises because some integrity constraints become violated. Also notice that weak executability is a necessary (but not sufficient) condition for strong executability. Then SE operations are a subset of WE operations.

In contrast with most related works, our method follows a *lightweight* approach. The term *lightweight* was popularized over almost twenty years ago following the publication of a round-table article [50], which invited the use of formal methods in a practical way. We consider our method is lightweight since it performs an static analysis of the model, i.e. it examines the model without executing its operations [34]. Static analysis has been mainly applied to analyze the source code of programs [48], but the same idea may be extended to analyze models at design time without requiring any kind of simulation. Besides, our method provides valuable information as feedback (in case of error, it points out the source of the error and assists the designer during their correction). As we will see later, our method relies on some assumptions regarding the input model and has some trade-offs required to enable our lightweight analysis.

The work reported here extends our previous works [45] and [46] on several directions: (1) We increase the expressiveness of the models our method can deal with (2) In order to align our work with the new UML standards, we now specify the operations by means of the Alf action language [40] provided by the OMG; (3) We re-design our previous methods [45, 46] to provide a unique and integrated method for verifying weak and strong executability properties; (4) In our previous work [46] we only provide the skeleton of the method as a black box, while in this work we elaborate in detail each step of the method; and (5) We provide a prototype tool that implements our method.

To the best of our knowledge, only few works have been devoted to verify fUML/Alf specifications [4, 8, 16, 28, 30, 33]. As suggested in [36, 21] and more specifically in [43], the verification of such specifications should be studied.

Paper Organization. The rest of the paper is structured as follows. Section 2 presents the motivation of our work. Section 3 introduces the basic concepts and the notation that will be used in the rest of the paper. Section 4 defines the concepts of *execution paths* and *executability.* Section 5 describes our method, Section 6 presents the prototype tool that implements it and Section 7 discusses about its pros and cons. Finally, Section 8 presents the related work and Section 9 summarizes with the relevant conclusions and further work.

2. Motivation

Executability is one of the most fundamental correctness properties for behavioural models. As we introduced, operations can be classified as *strongly executable* (SE), *weakly executable* (WE) or *non executable* (¬E).

The previous properties can also be characterized in terms of probabilities: ¬E operations generate a consistent state with probability 0, WE operations generate a consistent state with probability strictly greater than 0 and SE operations generate a consistent state with probability exactly 1. Then, WE operations may be viewed as *optimistic operations* since they apply the changes to the system state without ensuring that those changes will not break any constraint. They *hope* that they will not but an external mechanism is needed to check this and act accordingly (e.g. rolling back the changes if they indeed violate the constraints). Instead, SE operations may be viewed as *safe operations* (they will be successfully executed in any scenario).

Making sure that all operations are WE ensures a basic level of correctness. However, there might be cases (for instance, some critical and/or concurrent systems) in which WE operations are not sufficient. In these cases we believe it is also necessary to guarantee SE operations. Making sure that all operations are SE at design time ensures a full level of correctness, by facilitating a lot the development (and run-time efficiency) of the system: since we know that operations always leave the system in a consistent state, we can avoid checking at the end of each operation execution whether all constraints are satisfied which improves the efficiency of the system. Building (and executing) such integrity checking mechanism is an error-prone and time-consuming process that can be avoided when using our method to guarantee the executability of all operations.

Executability is not a new property, it has been studied, for instance, to verify declarative operations [12, 49] and model-to-model transformations [11, 47] (for more details, see the related work in Section 8). However, there is a lack of methods to check this property on executable models.

2.1. The need for Verification: writing executable operations is not easy

In general, we cannot assume that designers are able to easily write executable operations by themselves. To prove this fact, we did a small experiment with a total of one hundred students of a Software Engineering course at the Open University of Catalonia (UOC), all of them had a good background about modeling and programming. The students were provided with a UML class diagram (containing 14 classes, 10 associations and 32 integrity constraints) representing a car sharing system and they were asked to design four action-based operations. All operations were of low and medium complexity in terms of the number of actions they contained (an average of 6.5 actions each operation).

The results (see Figure 2) showed that, among all the operations designed by the students, 67% of them were ¬E, 33% of them were WE and only 26%

were also SE. Figure 3 shows the frequency of the typical errors found in the operations designed by the students.

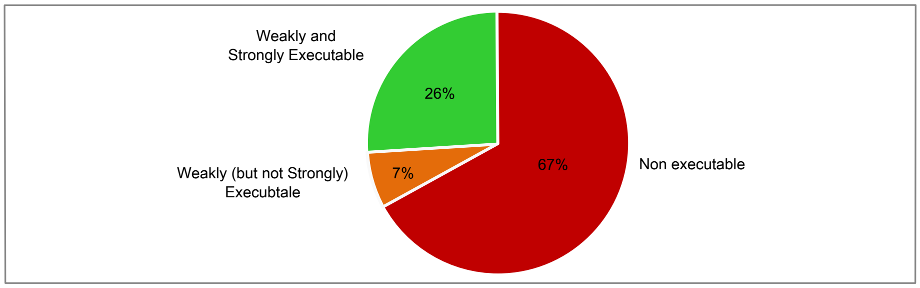


Figure 2: Results of our experiment.

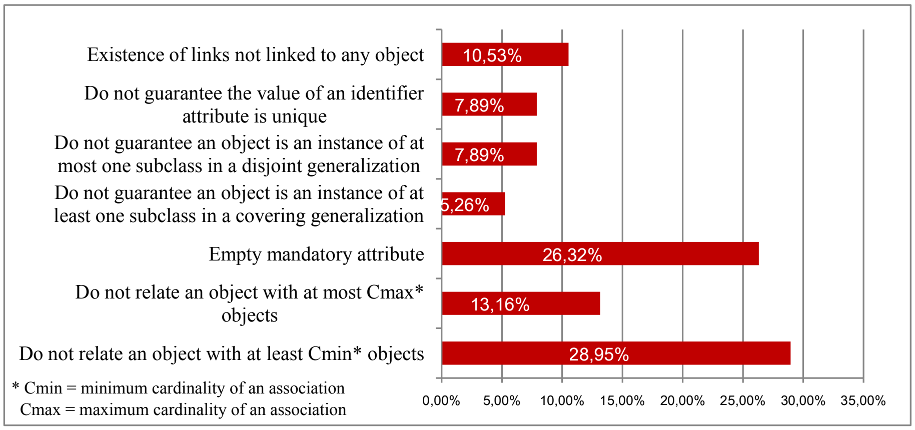


Figure 3: Frequency of the errors found in the operations of our experiment.

We believe this experiment supports the need for a method able to evaluate the correctness of the operations and that can be easily integrated in the modeling tools used by practitioners.

3. Basic Concepts and Notation

In order to facilitate the comprehension of our method, this section describes the basic concepts that will be used in the rest of the paper. In the following, the concepts of *executable model* (Section 3.1), *structural model* (Section 3.2) and *behavioural model* (Section 3.3) are reviewed.

In this paper we assume that executable models are written using the OMG standards (UML [39], OCL [38] and Alf [40] languages). Our choice is based on the wide adoption of the OMG standards in modeling specifications, although many concepts of this work are applicable to other languages as well.

3.1. Executable Model

It is widely accepted that a *model* is a simplified representation of a complex reality [10]. Moreover, an *executable model* is a model with a behavioural specification detailed enough so that it can be systematically implemented or executed in the production environment.

We represent an executable model (ExM) as a 2-tuple $\langle SM, BM \rangle$, where SM is a structural model and BM is a behavioural model.

3.2. Structural Model

A *structural model* specifies the static part of a software system, i.e. the general knowledge about the system domain [37].

We represent an structural model (SM) as a UML class diagram, composed by a set of *classes*, a set of *attributes* of each class, a set of *binary associations* and *generalizations* among classes and a set of *integrity constraints* (i.e. conditions that must be satisfied in all states of a software system [37]). Some integrity constraints (for instance, *cardinalities* and *disjointness/covering* constraints in generalizations) may be graphically represented in the class diagram, while others can be textually specified in OCL [38].

When verifying the executability of operations, our method takes into account the most commonly used integrity constraints according to [15]. They are shown in Table 1. First column (*Contraint*) identifies the constraint type, second column (*Abbreviation*) indicates the precise notation that our method internally uses, third column (*Description*) describes the meaning of the constraint, and last column (*Formalization in OCL*) provides its formal description. Some new constraints could be added to this table, however, as we discuss in Section 7 our method is not suitable to address arbitrary integrity constraints.

Table 1: Constraint types supported by our method.

Constraint	*Abbreviation*	*Description*	*Formalization in OCL*
Minimum cardinality of a class	**Cmin**(`cl`) = CminCl	Expresses the minimum objects of class `cl` that must exist simultaneously	**`context`** `cl` **`inv:`** `cl.allInstances() ->size() >= CminCl`
Maximum cardinality of a class	**Cmax**(`cl`) = CmaxCl	Expresses the maximum objects of class `cl` that may exist simultaneously	**`context`** `cl` **`inv:`** `cl.allInstances() ->size() <= CmaxCl`
Minimum cardinality of an association	**Cmin**(`as`,`r`) = CminAs	Expresses the minimum multiplicity of the member end (i.e. role) `r` of an association `as` between `cl` (with role `r`) and `cl'`	**`context`** `cl` **`inv:`** `cl.r->size() >= CminAs`
Maximum cardinality of an association	**Cmax**(`as`,`r`) = CmaxAs	Expresses the maximum multiplicity of the member end (i.e. role) `r` of an association `as` between `cl` (with role `r`) and `cl'`	**`context`** `cl` **`inv:`** `cl.r()->size() <= CmaxAs`
Mandatory attribute	**Mand**(`attr`, `cl`)	Expresses the attribute `attr` of class `cl` must have at least one value	**`context`** `cl` **`inv:`** `not cl.attr->oclIsUndefined()`

Continued on next page

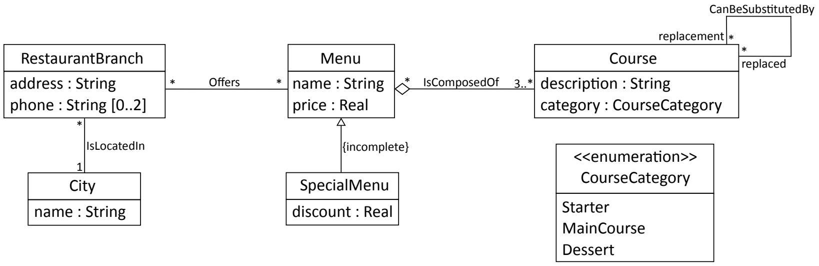
Table 1 – continued from previous page

Constraint	*Abbreviation*	*Description*	*Formalization in OCL*
Covering of a generalization	**Cov**(cl, $\{cl_1,\ldots,cl_n\}$) (cl generalizes $cl_1,\ldots,cl_n$)	Requires each instance of cl to be an instance of at least one cl_i.	**context** cl **inv:** self.oclIsTypeOf(cl_1) or ... or self.oclIsTypeOf(cl_n)
Disjointness of a generalization	**Disj**(cl, $\{cl_1,\ldots,cl_n\}$) (cl generalizes $cl_1,\ldots,cl_n$)	Requires each instance of cl to be instance of at most one cl_i	**context** cl **inv:** self.oclIsTypeOf(cl_i) implies not self.oclIsTypeOf(cl_x), where x,i=1..n and $x \neq i$.
Identifier	**ID**(attr,cl)	Expresses the attribute attr uniquely identifies instances of cl	**context** cl **inv:** cl.allInstances() -> isUnique(attr)
Symmetry of a recursive association	**Sym**(as)	Guarantees if an object o_1 is as-related to o_2, then o_2 is as-related to o_1	**context** cl **inv:** self.r -> forAll(o|o.r -> includes(self)), where r is a member end of as
Asymmetry of a recursive association	**Asym**(as)	Guarantees that if an object o_1 is as-related to o_2, then o_2 is not as-related to o_1	**context** cl **inv:** self.r -> forAll(o|o.r -> excludes(self)), where r is a member end of as
Irreflexivity of a recursive association	**Irrefl**(as)	Guarantees that an object o is never as-related to itself	**context** cl **inv:** self.r->excludes(self), where r is a member end of as
Value comparison	**ValueComp** (attr,op,v)	States a restriction on the value of the attribute attr: the expression attr op v (where op is a comparison operator and v is a value) must be true	**context** cl **inv:** self.attr <op> v
Referential integrity constraint	**Referential** (cl, as)	Guarantees each participant in the association as (in which cl participates) is an instance of its corresponding class	**context** cl **inv:** not self.r->oclIsUndefined(), where r is a member end of as

Example 1 As an example throughout this paper we will use the class diagram shown in Figure 4, meant to (partially) model the menus offered by a restaurant chain. The class diagram contains information about the restaurant branches, the menus they offer (the price of each menu is the same in all the branches) and the courses (at least three) that compose each menu. A course may have several substituting courses, which are suggested to the customer when the desired course is sold out. Discounts for special menus can be offered. In addition to the cardinalities included in the class diagram, on the bottom, several constraints are expressed in OCL (left) and using our abbreviated representation (right): (1) menuPrimaryKey states the name of a menu uniquely identifies[1] it; (2) atMost3SpecialMenus states there may exist at most three special menus simultaneously; (3) validDiscount states

[1] Although the latest version of UML [39] allows to represent the constraint ID, it does not define any associated notation. This is the reason why we represent this constraint by OCL.

the discount of a special menu must be at least 10 (per cent); (4) `symmetricAssociation` states the association `CanBeSubstitutedBy` is symmetric, i.e. if a course `c1` can be substituted by a course `c2` then `c2` can also be substituted by `c1`; and finally (5) `irreflexiveAssociation` guarantees that a course can not be substituted by itself.



context Menu **inv** menuPrimaryKey: Menu.allInstances()->isUnique(name)	ID(name,Menu)
context SpecialMenu **inv** atMost3SpecialMenus: SpecialMenu.allInstances()->size()<=3	Cmax(SpecialMenu)=3
context SpecialMenu **inv** validDiscount: self.discount >= 10	ValueComp(self.discount,>=,10)
context Course **inv** symmetricAssociation: self.replaced -> forAll(c|c.replaced -> includes(self))	Sym(CanBeSubstitutedBy)
context Course **inv** irreflexiveAssociation: self.replaced -> excludes(self)	Irrefl(CanBeSubstitutedBy)

Figure 4: Excerpt of a restaurant chain class diagram.

The *state* of a system at a specific time is the set of instances of the classes and associations defined in the class diagram that exist at that time [37]. It can be represented in UML using an object diagram.

Example 2 Given the class diagram of Figure 4, a possible system state called *currentState* would be a state in which there is a restaurant branch with address “Camèlies Street, 53, Barcelona”, which offers a non-special menu called “Anticrisis menu” for 5€ (see Figure 5) .

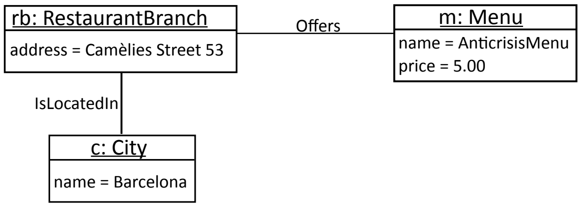


Figure 5: Object diagram representing the state *currentState* described in Example 2.

A state *s* **satisfies** an integrity constraint *ic* iff *ic* evaluates to true in this state. We denote by *Satisfies(s,ic)* the proposition that states that *s* satisfies the integrity constraint *ic*. Otherwise, we say that the constraint is *unsatisfied* or *violated.*

Let $ExM = \langle SM,\, BM \rangle$ be an executable model, a system state s is **consistent** regarding SM iff $\forall$ $ic \in SM$, *Satisfies(s,ic)*. We denote by *IsConsistent(s,SM)* the proposition that states that s is consistent regarding the structural model SM.

> **Example 3** The state *currentState* (see Figure 5) does not satisfy the minimum cardinality constraint of the association `IsComposedOf` in the role `course`, since the menu "Anticrisis menu" does not contain any course, i.e. *Satisfies(currentState, Cmin(*`IsComposedOf,course`*)=3) = false.*
>
> Therefore, this state is not consistent with the structural model of Figure 4, i.e. *IsConsistent(currentState, RestaurantChain_CD) = false.*

3.3. Behavioural Model

A *behavioural model* specifies the dynamic part of a software system, i.e. the valid changes in the system state, as well as the functions that the system can perform [37]. In UML there are several alternatives to represent the behaviour of a system but low-level specifications typically rely on the specification of operations (attached to UML classes) that are sequences of atomic steps that the users may execute to query/modify the information of the structural model. In this paper, we assume that a behavioural model (BM) is composed of a set of operations $\langle op_1,\ldots,\, op_n \rangle$, where each op_i is a sequence of Alf actions [40].

Alf is a standard published at the end of 2013 by the OMG (first beta version appeared in 2010). It provides a concrete syntax conforming to the fUML [41] abstract syntax, in charge of defining the basic read/write actions that can be used to specify the fine-grained behavioural aspects of systems plus some control flow statements to coordinate these actions in action sequences, conditional blocks or loops. The expressiveness of Alf is comparable to that of the instructions in traditional programming languages but at a higher abstraction (and platform-independent) level. We assume that operations are executed in an atomic transaction and they always terminate.

Table 2 shows the main modification actions provided by fUML (1st column), the corresponding concrete syntax provided by Alf (2nd column) and the description of the update they perform (3rd column). fUML predefines additional actions not shown in Table 2 since they do not affect our analysis.

> **Example 4** We show three operations (specified as UML activities in Alf) defined with Alf: (1) `newCourse` (in the context of class *Course*), creates a new course in the system; (2) `addMenu` (in the context of class *Menu*), adds a new menu to the system; and (3) `classifyAsSpecialMenu` (in the context of class *Menu*), classifies a menu as special menu.

[2] We assume that for every pair of objects there is at most one link in *as* between them.

Table 2: Main modification actions provided by fUML and concrete syntax in Alf.

fUML Action	*Alf Syntax*	*Description*
CreateObject	<object> = **new** <class>()	Creates and returns a new object of type class
DestroyObject	<object>.**destroy**()	Destroys the object object from its class and from any immediate superclasses (if such exists)
ReclassifyObject	**classify** <object> [**from** <oldCl>] [**to** <newCl>]	Removes object from classes in oldCl and/or adds it as a new instance of classes in newCl
AddStructuralFeature	<object>.<attribute> = <value>	Sets value as the new value for the attribute attribute of object object
ClearStructuralFeature	<object>.<attribute> = null	Removes all values for the attribute attribute of object object
CreateLink	<association>.**createLink** (<object1>,<object2>)	Creates a new link[2] (i.e. association instance) in the binary association association between object1 and object2
DestroyLink	<association>.**destroyLink** (<object1>,<object2>)	Destroys the link (i.e. association instance) in the binary association association between object1 and object2
ClearAssociation	<association>.**clearAssoc** (<object>)	Destroys all links of the named association that have at least one end with value object
OperationCall	[<result>]=<object>.<operation>([<arguments>])	Invokes the operation in the context of the object with arguments and, optionally, returns the result

```
activity newCourse(in _description:String, in
_substitutingCourses:Course[*]) {
  Course c = new Course();
  c.description = _description;
  for ( i in 1.._substitutingCourses→size() ) {
   CanBeSubstitutedBy.createLink(c,_substitutingCourses[i]);
  }
 }
```

```
activity addMenu(in _name:String, in _price:Real, in
_courses:Course[3..*]) {
  if ( !Menu.allInstances()→exists(m|m.name=_name) ) {
   Menu m = new Menu();
   m.name = _name;
   m.price = _price;
   for ( i in 1.._courses→size() ) {
    IsComposedOf.createLink(m,_courses[i]);
   }
  }
 }
```

```
activity classifyAsSpecialMenu(in _discount:Real) {
  if ( _discount ≥ 10 ) {
   classify self to SpecialMenu;
   self.discount = _discount;
  }
 }
```

4. Execution Paths and Executability

The aim of this section is to precisely define the notion of *executability* regarding our two levels of correctness (*weak* and *strong*). With this purpose we need to first introduce the concept of *execution path*.

4.1. Execution Paths

The verification of the executability property is based on an analysis of the possible *execution paths* allowed by the actions that define the operation effect, i.e., the possible sequences of actions that may be followed during its execution.

In order to determine the execution paths, we propose to draw each operation as a Model-Based Control Flow Graph (MBCFG), a directed graph based on the model specification (instead of on the program code, as traditional control flow graph proposals). MBCFGs have also been used to express UML Sequence Diagrams [20]. We adapt this idea to express the control flow of action-based operations.

In order to represent Alf operations as MBCFGs, we consider each operation is an instance of the *Activity* metaclass from fUML (see the fUML metamodel excerpt in Figure 6). An *Activity* contains several $ActivityNode$s ($ActivityNode$ generalizes the metaclass $ExecutableNode$, which in turn generalizes the metaclass *Action*). Then, each action can be either one of the actions from the Actions package (see them in Table 2), a $ConditionalNode$ (which represents an exclusive choice among some number of alternatives) or a $LoopNode$ (which represents a loop with setup, test, and body sections). We also use two $fake$ nodes: an *initial node* (representing the first instruction in the operation) and a *final node* (representing the last one). These two nodes do not change the operation effect but help in simplifying the representation of our MBCFG.

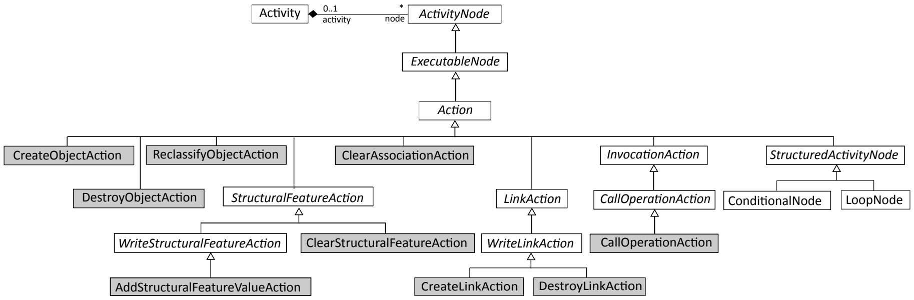


Figure 6: Fragment of the fUML metamodel Actions Package.

A Model-Based Control Flow Graph (MBCFG) for an operation *op* is a 2-tuple (V_{op}, A_{op}). The corresponding vertices (V_{op}) and arcs (A_{op}) are obtained applying the following rules:

- Every activity node (i.e. action) in *op* is a vertex in V_{op}. In order to simplify the MBCFG, we only consider actions that may modify the system state (i.e. modification actions) and structured actions (i.e. conditionals or loops). It means that we skip other types of actions (as actions to read values or to declare and initialize variables) since they do not affect the result of our analysis.
- An arc from a vertex v_1 to v_2 is created in A_{op} if v_1 immediately precedes v_2 in an ordered sequence of nodes.

- A vertex v representing a conditional node n is linked by an arc to the vertices v_1, …, v_n representing the first activity node for each *clause* (i.e. the *then* clause, the *else* clause, …) in n. All vertices of each clause are englobed into a dashed line box. The last vertex in each clause is linked to the vertex v_{next} immediately following n in the sequence of executable activities. If n does not include an *else* clause, an arc between v and v_{next} is also added to A_{op}.
- Each arc from a conditional node to its first clause vertex is labelled with the condition of the conditional structure. Each arc to an *else* clause (or the arc between the conditional node and the the v_{next} if there is not an *else* clause) is labelled with the negation of the above condition.
- A vertex v representing a loop node n, is linked by an arc to the vertex representing the first activity node for $n.bodyPart$ (returning the list of actions in the body of the loop) and the vertex v_{next} immediately following n in the activity. The last vertex in $n.bodyPart$ is linked back to v (to represent the loop behaviour).
- Each arc from a loop node to its first vertex is labelled with the condition fulfilled in the first execution of the loop followed by the times the loop is executed.
- A vertex representing an *OperationCall* action is replaced by the subdigraph corresponding to the called operation op' as follows: (1) the initial vertex of op' is connected with the vertex that precedes the *OperationCall* activity node in the main operation; (2) the final vertex of op' is connected with the vertex/ces that follow the *OperationCall*; and (3) the parameters of op' are replaced by the arguments in the call.

Example 5 Figures 7, 8 and 9 show the MBCFGs for the operations `newCourse`, `addMenu` and `classifyAsSpecialMenu` respectively, where each node has been labelled with a number.

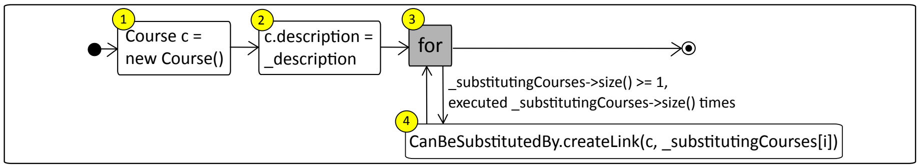

1
Course c = new Course()
2
c.description = _description
3
for
_substitutingCourses->size() >= 1,
executed _substitutingCourses->size() times
4
CanBeSubstitutedBy.createLink(c, _substitutingCourses[i])

Figure 7: MBCFG of `newCourse` operation.

The process to verify the executability is based on an analysis of the possible *execution paths* allowed by the MBCFG. An execution path of an operation *op* is a finite and not empty sequence of actions that may be followed during the operation execution (note that empty paths are discarded). For trivial operations (e.g. with neither conditional nor loop nodes) there is a single execution path but, in general, several ones will exist.

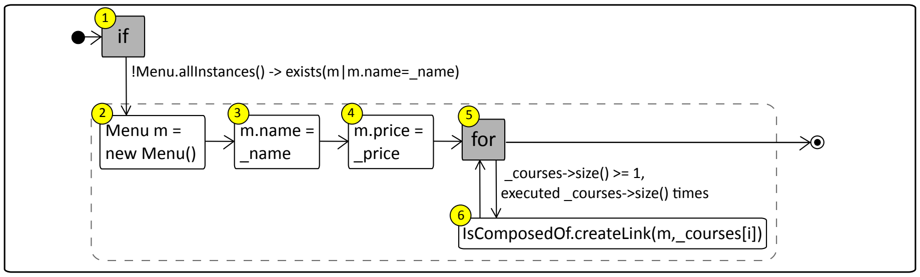

1
if
!Menu.allInstances() -> exists(m|m.name=_name)
2
Menu m =
new Menu()
3
m.name =
_name
4
m.price =
_price
5
for
_courses->size() >= 1,
executed _courses->size() times
6
IsComposedOf.createLink(m,_courses[i])

Figure 8: MBCFG of `addMenu` operation.

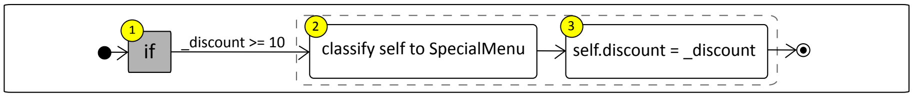


Figure 9: MBCFG of `classifyAsSpecialMenu` operation.

Given a $MBCFG_{op}$ for an operation *op*, the set of execution paths ($Paths(op)$) for *op* is defined as $Paths(op) = allPaths(MBCFG_{op})$, where $allPaths(MBCFG_{op})$ returns the set of all paths in $MBCFG_{op}$ that start at the initial vertex (the vertex corresponding to the initial node), end at the final vertex and does not include repeated arcs.

Example 6 Operations `newCourse` and `addMenu` have two execution paths (see Figures 10 and 11 respectively) and operation `classifyAsSpecialMenu` has a single execution path (see Figure 12).

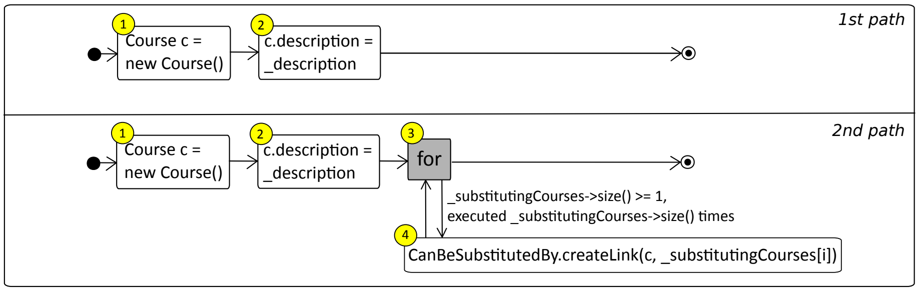

1st path
1
Course c =
new Course()
2
c.description =
_description
2nd path
1
Course c =
new Course()
2
c.description =
_description
3
for
_substitutingCourses->size() >= 1,
executed _substitutingCourses->size() times
4
CanBeSubstitutedBy.createLink(c, _substitutingCourses[i])

Figure 10: Paths of `newCourse` operation.

4.2. Executability

The execution of an execution path *p* over a system state *s*, generates a new state *s'* where the changes described in *p* have been applied to *s*. We denote by *AllExecutions(p,s)* $= \{s'_1,\ldots,s'_n\}$ all the possible (potentially infinite)

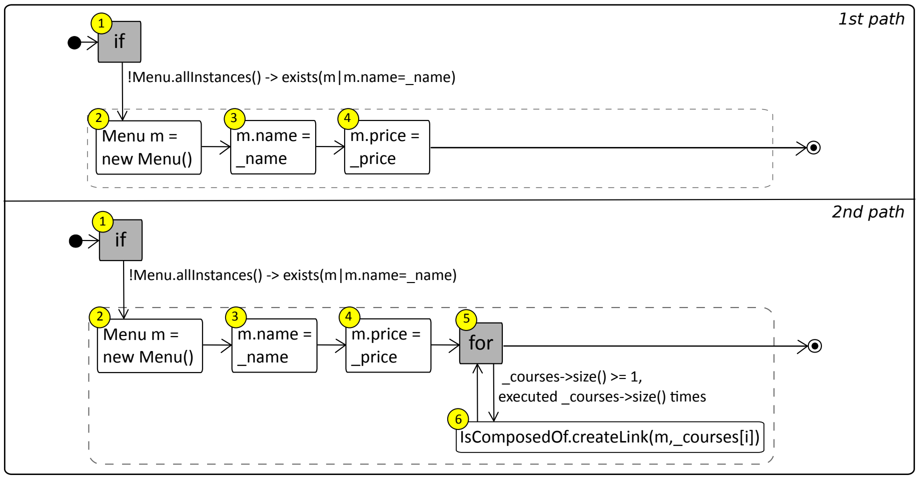

1st path
1
if
!Menu.allInstances() -> exists(m|m.name=_name)
2
Menu m =
new Menu()
3
m.name =
_name
4
m.price =
_price
2nd path
1
if
!Menu.allInstances() -> exists(m|m.name=_name)
2
Menu m =
new Menu()
3
m.name =
_name
4
m.price =
_price
5
for
_courses->size() >= 1,
executed _courses->size() times
6
IsComposedOf.createLink(m,_courses[i])

Figure 11: Paths paths of `addMenu` operation.

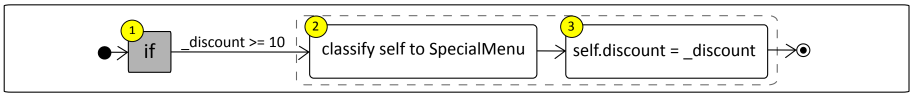


Figure 12: Unique path of `classifyAsSpecialMenu` operation.

executions of the execution path p over a system state s, that is, all the possible states $(s'_1,\ldots,s'_n)$ that may be reached (depending on the input arguments used to call the operation) by executing p in s.

4.2.1. Weakly Executable operations

We consider an operation is *weakly executable* (WE) if it may generate a consistent state, but it is not guaranteed to do so. That is to say, an operation is WE if there is a chance that a user may successfully execute it, i.e. if exists at least an initial state and a set of arguments for the operation parameters for which the execution of the actions included in the operation evolves the initial state of the system to a new state that satisfies all the integrity constraints of the structural model. Note that *weak executability* does not require the success of all executions of the operation to be successful.

More formally:

Let ExM = ⟨SM,BM⟩ be an executable model, an operation op ∈ BM is **weakly executable** (WE) iff ∃ p ∈ Paths(op) ∧ ∃ s IsConsistent(s,SM) ∧ ∃ s' ∈ AllExecutions(p,s) | IsConsistent(s',SM)

Example 7 Operation `newCourse` is not WE since it never may generate a consistent system state regarding the structural model of Figure 4: every time we try to create a new course `c` but we do not assign any category for it, we reach an inconsistent system state where c has no category, a situation forbidden by the structural model that defines the attribute `category` as mandatory ($Mand$(`category`,`Course`)), i.e. it must have at least one value. Instead, operation `classifyAsSpecialMenu` is WE since we are able to find an execution scenario (a system state that contains less than three special menus) where the menu can be successfully subtyped. Note that classifying an operation as WE does not mean that every time this operation is executed the new system state will be consistent with the integrity constraints. For instance, if the system state where we apply the `classifyAsSpecialMenu` operation already contains three special menus, the operation will fail because of the integrity constraint $Cmax$(`SpecialMenu`)=3, which defines the class `SpecialMenu` may have at most three instances.

4.2.2. Strongly Executable operations

We consider an operation is *strongly executable* (SE) if it is guaranteed to always generate a consistent state. That is to say, 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 state that satisfies all the integrity constraints of the structural model. Note that, unlike *weak executability*, *strong executability* requires all executions of the operation to be successful.

More formally:

Let ExM = ⟨SM,BM⟩ be an executable model, an operation op ∈ BM is **strongly executable** (SE) iff ∀ p ∈ Paths(op) ∧ ∀ s IsConsistent(s,SM) ∧ ∀ s' ∈ AllExecutions(p,s) IsConsistent(s',SM)

Example 8 As we have seen, operation `classifyAsSpecialMenu` is not SE since after its execution we may violate the maximum cardinality integrity constraint of class `SpecialMenu` ($Cmax$(`SpecialMenu`)=3), in particular, when the system state where the operation is applied already contains three special menus. Instead, operation `addMenu` is SE since we may guarantee it will never violate any integrity constraint of the structural model after their execution. Note that, in this case, the operation `addMenu` includes a guard (i.e. a precondition) to guarantee the changes this operation performs will only be executed in a safe context.

4.3. Non Executable operations

Operations that are not WE are non executable (¬E). *Non executable* operations never generate a consistent system state. After their execution, they always reach a state of the system that violates some integrity constraints of the structural model (e.g. some cardinality constraints).

More formally:

Let ExM = ⟨SM,BM⟩ be an executable model, an operation op $\in$ BM is **non executable** (¬E) iff $\forall$ p $\in$ Paths(op) $\wedge$ $\forall$ s IsConsistent(s,SM) $\wedge$ $\forall$ s' $\in$ AllExecutions(p,s) ¬IsConsistent(s',SM)

Example 9 Since `newCourse` is not WE, it is non executable.

5. Verifying Executable Models

In the previous section we have intuitively seen examples of executable and non-executable operations (see Examples 7-9). However, for general and complex operations, the manual reasoning to verify this property is tedious and error-prone. In this section we provide a lightweight and static method that automatizes this process to help the designers to verify her executable models.

Our method (see Figure 13) takes as input an executable model composed by a structural model (a UML class diagram) and an Alf-based operation (as part of the behavioural model). Then, our method returns either a positive answer, meaning that the operation is WE/SE or a corrective feedback (see Section 5.4), consisting in a set of actions and guards that should be added to the operation in order to make it WE/SE. Note that, extending the operation with the provided feedback is a necessary condition but not a sufficient one to immediately guarantee the WE/SE of the operation since the added actions may in its turn induce other constraint violations. Therefore, the extended operation must be iteratively reanalyzed with our method until we reach a WE/SE status.

When analyzing the WE/SE of an operation we must take into account all the possible *execution paths* (see Section 4): an operation is WE iff at least one of its execution paths is WE; and it is SE iff all its executions paths are SE; otherwise it is ¬E. Therefore, prior to checking the weak/strong executability of an operation, our method performs a pre-processing step to compute its execution paths (Step 0). Once the execution paths have been computed, Steps 1 and 2 of the method are applied on each path *p* until we recognize a WE path (in case of verifying weak executability) or until we check all paths are SE (in case of verifying strong executability). First, Step 1 (see Section 5.1) individually analyzes each action in the path *p* to see whether it may violate some integrity constraints of the structural model. Then, Step 2 (see Section 5.2) performs a contextual analysis of each potentially violating action to see whether other actions or conditions in *p* compensate or complement its effect to ensure that we sometimes/always reach a consistent state at the end of the operation execution. If all potentially violating actions can be discarded we can conclude that *p* is WE/SE. Finally, Step 3 (see Section 5.3) classifies the operation depending on the results obtained in the previous step.

Our method performs an over-approximation analysis. Over-approximation is due to the lack of exhaustiveness in the comparison of conditions in the operation to favor the efficiency of the process. This implies that our method may return *false positives*, that is, it may return as a non WE/SE an operation which is actually WE/SE. On the other hand, the method does not return

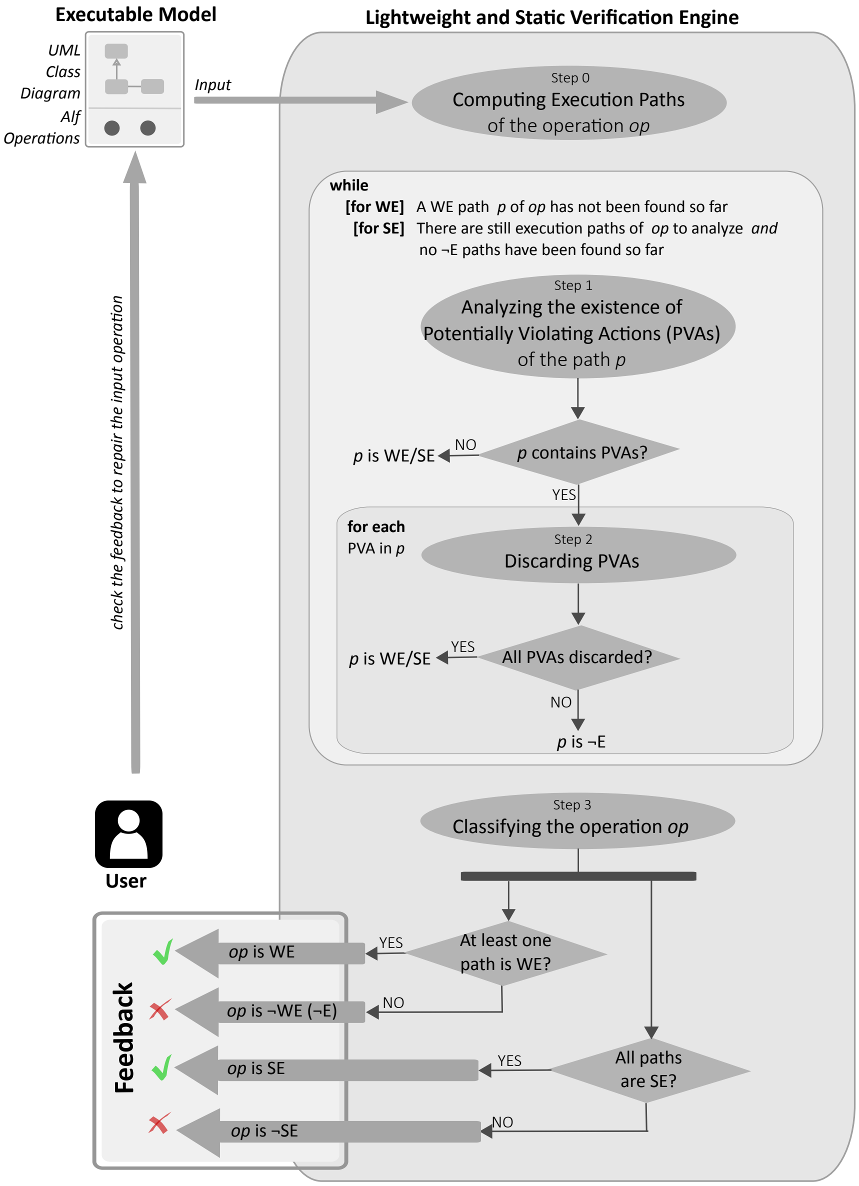

Executable Model
UML
Class
Diagram
Alf
Operations
Input
Lightweight and Static Verification Engine
Step 0
Computing Execution Paths
of the operation op
while
[for WE] A WE path p of op has not been found so far
[for SE] There are still execution paths of op to analyze and
no ¬E paths have been found so far
Step 1
Analyzing the existence of
Potentially Violating Actions (PVAs)
of the path p
p is WE/SE
NO
p contains PVAs?
YES
for each
PVA in p
Step 2
Discarding PVAs
p is WE/SE
YES
All PVAs discarded?
NO
p is ¬E
Step 3
Classifying the operation op
check the feedback to repair the input operation
User
Feedback
op is WE
YES
At least one
path is WE?
op is ¬WE (¬E)
NO
op is SE
YES
All paths
are SE?
op is ¬SE
NO

Figure 13: Method overview.

false negatives (in our opinion, more critical than the above), that is, when it states that an operation is WE/SE, this statement is always true. As will be explained in Section 7, this over-approximation may be manually eliminated by the designer if she decides to intervene during the second step of the method.

In the following subsections we describe the three steps of our verification method (Sections 5.1, 5.2 and 5.3) and the feedback it provides (Section 5.4).

5.1. Step 1: Analyzing the existence of Potentially Violating Actions

First step of our verification method analyzes each action in the path to see whether its effect can change the system state in a way that some integrity constraints of the structural model become violated. If so, this action is declared as *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, it is WE/SE (and if we are checking if the operation is WE, we can directly confirm it at this step). Otherwise, we need to continue the analysis with the next step.

In order to detect the PVAs we have defined a set of rules that automatically determine the actions that may violate each integrity constraint of the structural model. Table 3 shows these rules. First column (*Susceptible Violated Constraint (SVC)*) shows each constraint our method supports (see them at Section 3) and second column (*Potentially Violation Actions (PVAs)*) determines the modification actions each constraint may violate. Several subrows for the same integrity constraint indicate several actions that may violate this constraint. Sharp sign (#) represents irrelevant variables and consecutive letters (x, y,...) represent free variables that may be bound to any value in the action. Note that, when the minimum cardinality constraint of a class (*Cmin*(cl)) or of an association (*Cmin*(as,role)) is not restricted (i.e. it is equal to zero), then no action may violate this constraint. Similarly, when the maximum cardinality constraint of a class (*Cmax*(cl)) or of an association (*Cmax*(as,role)) is not restricted (i.e. it is equal to "*"), then no action may violate this constraint.

Table 3: Rules to determine the actions that may violate each integrity constraint.

	Susceptible Violated Constraint (SVC)	*Potentially Violating Actions (PVAs)*
1	Cmin(cl)≠0	o.**destroy**(), where o is an instance of class cl or of a subclass of cl
		classify x **from** oldCl, where oldCl includes the class cl or one of its subclasses (only applies when cl is child of a generalization)
2	Cmax(cl)≠*	x = **new** cl()
		x = **new** cl′(), where cl′ is a subclass of cl
		classify x **to** newCl, where newCl includes the class cl or one of its subclasses (only applies when cl is child of a generalization)
3	Mand(attr,cl)	x = **new** cl()
		x = **new** cl′(), where cl′ is a subclass of cl
		classify x **to** newCl, where newCl includes the class cl or one of its subclasses (only applies when cl is child of a generalization)
		x.attr = null, where x.oclIsTypeOf(cl) or x.oclIsTypeOf(cl′) and cl′ is a subclass of cl
4	Cmin(as,r)≠0	x = **new** cl(), where cl (or one of its superclasses) participates on the association as with role r′ (r′ is the opposite role to r in as)
		classify x **to** newCl, where newCl includes the class cl and cl (or one of its superclasses) participates on the association as with role r′ (r′ is the opposite role to r in as) (only applies when cl is child of a generalization)
		as.**destroyLink**(x,y), where the pair of objects (x,y) participate on the association as
		as.**clearAssoc**(o), where o participates on the association as with role r′ (and r′ is the opposite role to r in as)

Continued on next page

Table 3 – continued from previous page

	Susceptible Violated Constraint (SVC)	*Potentially Violating Actions (PVAs)*
5	Cmax(as,r)$\neq$*	as.**createLink**(x,y)
6	Cov(cl,$\{cl_1$,..., $cl_n\}$)	**classify** x **from** oldCl, where oldCl includes one cl_i
7	Disj(cl,$\{cl_1$,..., $cl_n\}$)	**classify** x **to** newCl, where newCl includes one cl_i
8	ID(attr,cl)	o.attr = #, where o is an instance of the class cl or of a subclass of cl
9	Sym(as)	as.**createLink**(x,y)
		as.**destroyLink**(x,y)
10	Asym(as)	as.**createLink**(x,y)
11	Irrefl(as)	as.**createLink**(x,x)
12	ValueComp (attr,op,v)	o.attr = #, where o is an instance of the class which owns attr or of one of its subclasses
13	Referential(cl,as)[4]	**classify** o **from** oldCl, where o.oclIsTypeOf(cl) (before classifying o), oldCl includes the class cl and cl participates on the association as (only applies when cl is child of a generalization)

As an example, we discuss the first row of Table 3, which determines the actions that may violate the minimum cardinality constraint of a class *cl* when it is different to zero ($Cmin$(cl)$\neq$0):

- First subrow indicates every time we destroy an object of class cl or of a subclass of cl (that is, the number of instances of cl is decreased), we may violate the constraint $Cmin$(cl).
- Similarly, second subrow indicates every time we take out an object from class cl or from one of its subclasses, we also may violate this constraint.

In this first step, the rules of Table 3 are applied over all the integrity constraints that appear in the input structural model. As a result, we obtain the set of potentially violating actions (PVAs) that may violate each integrity constraint of the structural model. Then, we may determine whether a path p contains PVAs by comparing this set of actions with the set of actions which appear in p. All actions in the intersection of both sets are PVAs.

In order to do this comparison a mapping between the PVAs obtained from Table 3 and the actions of the path has to be done. An action of the first set (cointaining generic PVAs) can be mapped onto an action of the second set (containing specific PVAs obtained from the operation paths) when the following conditions are satisfied: (1) both actions are from the same type (e.g. *CreateObject*, $ReclassifyObject$, etc.); (2) the model elements referenced by the actions coincide (e.g. both *CreateObject*s create objects of the same class); and (3) all instance-level parameters of the generic PVA (i.e. variables x, y,...)

[3] Note that this constraint is not violated when we destroy an object of type cl, because the Alf semantics for the action DestroyObject ensures the destruction of all links in which the destroyed object participates.

can be bound to the parameters of the specific PVA (irrelevant variables - i.e. # - may be bound to any parameter value in the specific PVA).

Example 10 As an example, we show the PVAs for the two execution paths of operation `newCourse`. Second path (see Figure 14, where PVAs are highlighted in red) contains two PVAs: (1) the 1st action in the path, which may violate two mandatory constraints (when the attributes *description* and *category* are not initialized); and (2) the last action in the path, which may violate the *symmetric association* constraint (when the opposite link is not created) and the *irreflexive association* constraint (when the link connects an object with itself). First path (which is a subset of the former) only contains the first PVA. Then, in order to determine if these paths are WE/SE, we need to continue the analysis with the next step.

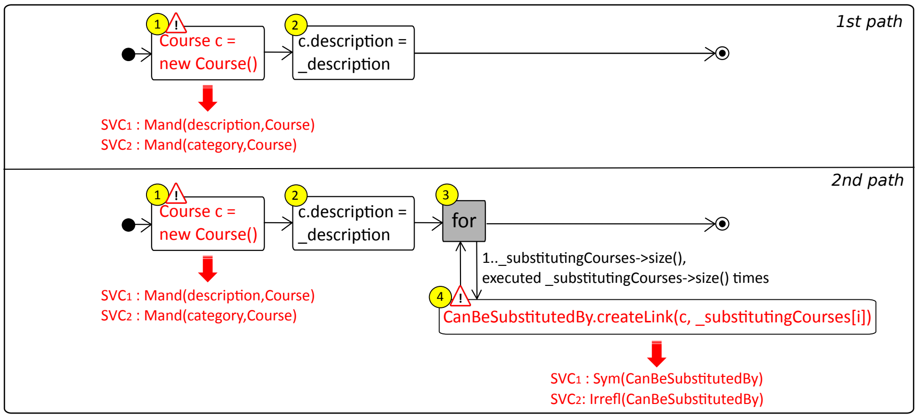

1st path
1
Course c =
new Course()
2
c.description =
_description
SVC$_1$: Mand(description,Course)
SVC$_2$: Mand(category,Course)
2nd path
1
Course c =
new Course()
2
c.description =
_description
3
for
1.._substitutingCourses->size(),
executed _substitutingCourses->size() times
4
CanBeSubstitutedBy.createLink(c, _substitutingCourses[i])
SVC$_1$: Mand(description,Course)
SVC$_2$: Mand(category,Course)
SVC$_1$: Sym(CanBeSubstitutedBy)
SVC$_2$: Irrefl(CanBeSubstitutedBy)

Figure 14: PVAs of the paths of the operation `newCourse`.

5.2. Step 2: 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 not going to actually violate any of its SVCs. In these cases, the PVA may be discarded. Roughly, there are two ways to discard a PVA: (1) when the path contains a guard (i.e. a precondition) 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 after executing the operation.

In this second step, our method analyzes each PVA returned by the previous step and tries to discard them by analyzing the above possibilities. If all PVAs that may compromise the WE/SE of the path can be discarded, then it is classified as WE/SE. If not, the path is marked as ¬WE/¬SE and the corresponding corrective feedback is provided (see Section 5.4). Note that, if a

PVA may violate several SVCs, it may be discarded only when it satisfies all the conditions to avoid violating each SVC.

The conditions to discard the PVAs are expressed as action patterns that should be matched in the path. To the sake of simplicity, here we only show a small subset of the forty patterns we designed (the complete catalog of patterns can be found on [44]). They are shown in Table 4, which has several columns:

- *PVA*: States the PVA we are trying to discard.
- *SVC*: States the constraint the PVA may violate.
- *Conditions to discard the PVA*: Describes the conditions the path must satisfy to discard the PVA in order to avoid the violation of the SVC. Conditions are expressed as an *Alf pattern*, i.e. in reference of Alf statements that the path should or should not include complemented with a textual *description*.

Note that when trying to match guard expressions (for instance, conditional structures) we follow a syntactic approach, i.e. we do not try to formally prove the expression in the path implies the expression in the guard (which would be too costly for general expressions) but just to check whether a path expression matches one of the syntactic variation patterns predefined for the condition. For instance, if our pattern is expecting the guard `cl.allInstances()->size()<x` (where `x` is an integer) and the checked operation contains the guard `cl.allInstances()->size()<OCLexpr` (where `OCLexpr` is a new OCL expression instead an integer), our algorithm does not semantically compares `x` and `OCLexpr` but concludes the two guards do not syntactically match. When the algorithm cannot conclude the implication it assumes that one does not imply the other. This is why the method over-approximates the results (as a necessary trade-off to foster the efficiency of the method) as commented previously. The designer could optionally participate in this step to manually identify those implications that were not found by the method using its syntactic approach.

Table 4: Conditions to discard PVAs.

	PVA	**SVC**	**Conditions to discard the PVA**	

1	`o = new cl()`	Mand(`attr`,`cl`)	*Pattern*	`o = new cl(); //PVA`
				`...`
				`o.attr = #;`

			Descr.	The path includes, after the PVA, at least one action to initialize the attribute *attr*.

2	`o = new cl()`	Cmin(`as`,`r`)≠0	*Pattern*	`o = new cl(); //PVA`
				`...`
				`for ( i in 1..≥Cmin(as,r) ) {`
				` as.createLink(o,`x_i`); //`x_i` participates in as with role r`
				`...`
				`}`

			Descr.	The path includes, after the PVA, at least Cmin(`as`,`r`) actions to create a link of `as` between the new object `o` and another object (with role `r`).
				Continued on next page

Table 4 – continued from previous page

	PVA	*SVC*	*Conditions to discard the PVA*	

3	as.createLink(o_1,o_2)	Sym(as)	*Pattern*	as.createLink(o_1,o_2); //PVA
				...
				as.createLink(o_2,o_1);

			Descr.	The path includes the creation of the symmetric link.

		Irrefl(as)	*Pattern*	if (o_1 $\neq$o$_2$) {
				as.createLink(o_1,o_2); //PVA
				...
				}

			Descr.	The path contains a guard that prevents the execution of the PVA when the two member ends are the same object.

Example 11 As an example, we try to discard the PVAs of the second execution path of the operation newCourse (note that the first path is a subset of the former). As we justified in the previous step, this path contains two PVAs: (1) the first action: Course c = new Course() and (2) the last: CanBeSubstitutedBy.createLink(c,_substitutingCourses[i]). According to 1st row of Table 4, in order to discard the first PVA (see action 1 of Figure 15) when it may violate a constraint of type *mandatory*, the path must include, after the PVA, at least one action to initialize the attributes description and category. The path contains an action (see action 2 of Figure 15) to initialize the attribute description but it does not contain any action to initialize the attribute category. Then the first PVA cannot be discarded because it always violate the constraint $Mand$(category,Course).

At this point, our method can conclude this path is ¬E since it always violates the above constraint. However, in order to illustrate a complete example, in the following we analyze the remaining PVAs.

According to 3rd row of Table 4 (1st subrow), in order to discard the second PVA (see action 4 of Figure 15) when it may violate a constraint of type *symmetric*, the path must include the creation of the symmetric link. The path does not create this link. Besides, according to 3rd row of Table 4 (2nd subrow), in order to discard the same PVA when it may violate a constraint of type *irreflexive*, the path must include a guard to prevent the execution of the PVA when the two courses are the same object. The path does not include this guard. Then the second PVA cannot be discarded and, as we pointed before, our method concludes this path is not WE/SE.

5.3. Step 3: Classifying the operation

Last step of our method classifies the operation depending on the results obtained in the previous step regarding each execution path of the operation.

If at least one of the execution paths of the operation is WE, the operation is classified as WE. If all its execution paths are SE, the operation is classified as SE. Otherwise, the operation is classified as ¬E.

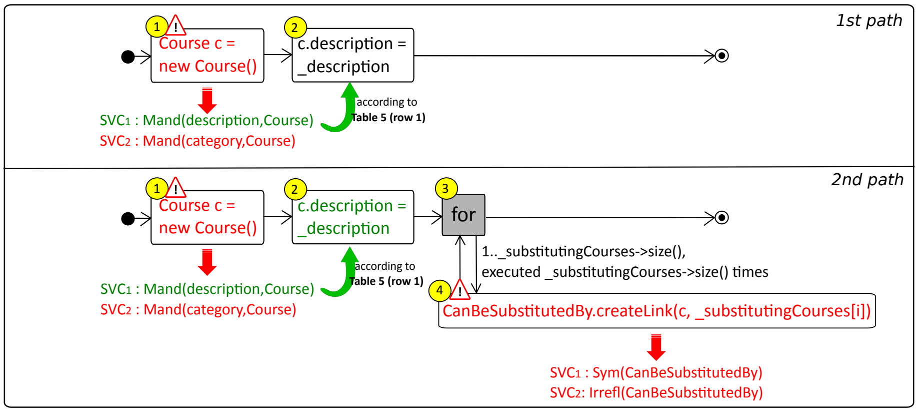

1st path
1
Course c = new Course()
2
c.description = _description
SVC₁ : Mand(description,Course)
SVC₂ : Mand(category,Course)
according to Table 5 (row 1)
2nd path
1
Course c = new Course()
2
c.description = _description
3
for
1.._substitutingCourses->size(), executed _substitutingCourses->size() times
SVC₁ : Mand(description,Course)
SVC₂ : Mand(category,Course)
according to Table 5 (row 1)
4
CanBeSubstitutedBy.createLink(c, _substitutingCourses[i])
SVC₁ : Sym(CanBeSubstitutedBy)
SVC₂: Irrefl(CanBeSubstitutedBy)

Figure 15: Discarding PVAs for the second path of `newCourse` operation.

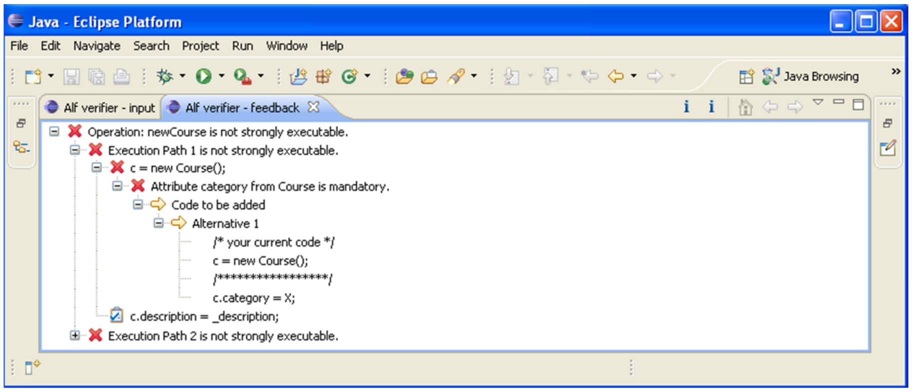
Example 12 Since, as shown previously, both paths of `newCourse` operation are not WE neither SE, our method concludes this operation is ¬E. Next section shows how to correct this.

5.4. Feedback

Besides determining the executability of an operation, a distinguishing feature of our method is that for ¬WE/¬SE operations it returns valuable information to help the designers identifying and correcting the detected errors. This feedback information is expressed in terms of the operation itself so it can be easily understood and processed by the designer.

For ¬WE/¬SE operations, our method provides two kinds of information: (1) the returned feedback identifies *why* the operation is non executable, i.e. for each ¬WE/¬SE path our method provides the list of PVAs that could eventually induce a violation of the integrity constraints together with the specific list of SVCs that those PVAs could violate; and (2) the returned feedback explains *how* the designer may fix these (potentially) violating scenarios by providing a set of possible repair alternatives that should be included in the ¬WE/¬SE operation paths. These alternatives are expressed as a finite set of Alf-patterns (see the complete catalog on [44]) to be added to the path of those PVAs that cannot be discarded. The designer should choose the most appropriate alternative in her context.

Example 13 Figure 16 shows the feedback provided when verifying the strong executability for the operation `newCourse`. Next, we show the repaired operation once the feedback provided by our method has been integrated. The added sentences are emphasized in bold type. Each added sentence fixes one of the problems detected in the previous section.



```
Operation: newCourse is not strongly executable.
  Execution Path 1 is not strongly executable.
    c = new Course();
      Attribute category from Course is mandatory.
        Code to be added
          Alternative 1
              /* your current code */
              c = new Course();
              /*****************/
              c.category = X;
    c.description = _description;
  Execution Path 2 is not strongly executable.
```

Figure 16: Feedback for `newCourse` operation.

```
activity newCourse(in _description: String, in _substitutingCourses:Course[*], in
_category: CourseCategory) {
   Course c = new Course();
   c.description = _description;
   c.category = _category;
   for ( i in 1.._substitutingCourses→size() ) {
     if ( c ≠ _substitutingCourses[i] ) {
       CanBeSubstitutedBy.createLink(c,_substitutingCourses[i]);
       CanBeSubstitutedBy.createLink(_substitutingCourses[i],c);
     }
   }
}
```

The initialization of the attribute `category` ensures the constraint $Mand$(`category,Course`) will never be violated and the new created link ensures the constraint Sym(`CanBeSubstitutedBy`) will never be violated. After applying these changes, the operation `newCourse` becomes WE and SE.

6. Tool Support

In order to prove the feasibility of our method, we have built a prototype tool that implements our algorithm for verifying Alf operations. The tool has been implemented as an Eclipse plug-in which can be downloaded from [1]. A demonstration video about it can be viewed on [2].

Figure 17 shows the general view of the tool architecture. As a first step, the designer specifies the UML executable model she wants to deal with. The structural model, composed by a UML class diagram and a set of OCL integrity constraints, is modelled using the graphical modelling environment provided by UML2Tools [52], an Eclipse Graphical Modeling Framework for manipulating UML models. The behavioural model, composed by a set of Alf operations, is specified in a text file with *.alf* extension. Once both models have been defined, the designer selects the operation/s and the property (weak or strong executability) she wants to verify. Then, the core of our method is invoked to

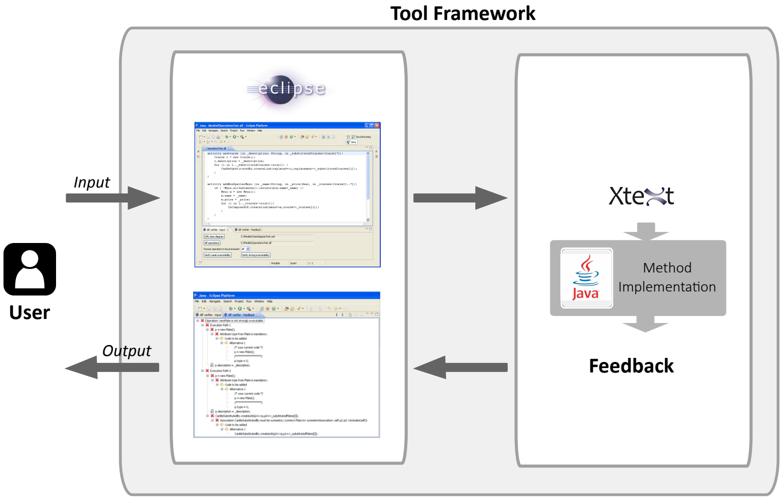


Figure 17: General architecture or our tool.

perform the static analysis we have described in Section 5. Finally, the feedback provided by our method is displayed, integrated into the Eclipse interface.

Our tool is implemented as a set of Java classes extended with the libraries of the UML2Tools (to interact with the input UML model) and Xtext [3] (to parse the Alf operations and instantiate them as Java classes).

As an example, Figure 18 shows the main form of our Eclispe plug-in, which permits to import both the UML model and the Alf operations and to choose operation/s and the property the designer wants to verify. A screenshot of the feedback for the operation `newCourse` provided by our method has been shown in Figure 16.

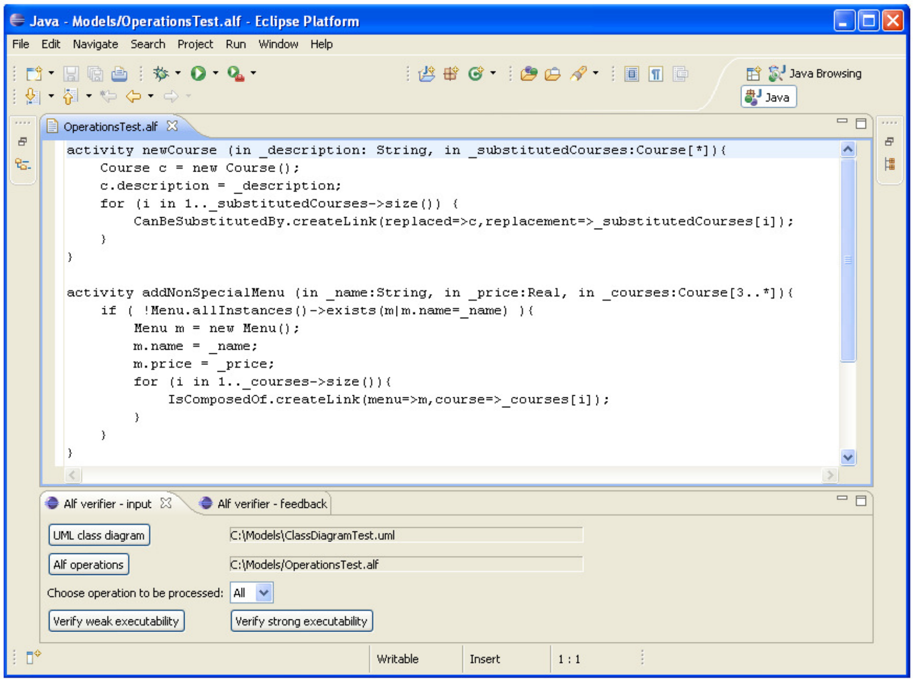
7. Discussion

In this section we expose the assumptions our method relies on and discuss their limitations in order to evaluate its pros and cons.

7.1. Assumptions and limitations of our method

Our method assumes all Alf operations are syntactically correct (i.e. they conform to the standard Alf language [40]) and terminate. This is a reasonable assumption and necessary to begin our analysis.

According to the widely accepted criteria about the elimination of the unreachable code [14], our method also assumes the body of all conditional and loop structures is reachable (given the proper input values). This means that the condition of all conditional and loop structures may be satisfied (i.e. they can evaluate to true) and then the body of these structures may be executed. Otherwise, the actions in those paths that may be needed to compensate the


Java - Models/OperationsTest.alf - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Java Browsing
Java
OperationsTest.alf
Alf verifier - input
Alf verifier - feedback
UML class diagram
Alf operations
Verify weak executability
Verify strong executability
Writable
Insert
1 : 1

```
activity newCourse (in _description: String, in _substitutedCourses:Course[*]){
    Course c = new Course();
    c.description = _description;
    for (i in 1.._substitutedCourses->size()) {
        CanBeSubstitutedBy.createLink(replaced=>c,replacement=>_substitutedCourses[i]);
    }
}

activity addNonSpecialMenu (in _name:String, in _price:Real, in _courses:Course[3..*]){
    if ( !Menu.allInstances()->exists(m|m.name=_name) ){
        Menu m = new Menu();
        m.name = _name;
        m.price = _price;
        for (i in 1.._courses->size()){
            IsComposedOf.createLink(menu=>m,course=>_courses[i]);
        }
    }
}
```

UML class diagram: C:\Models\ClassDiagramTest.uml

Alf operations: C:\Models\OperationsTest.alf

Choose operation to be processed: All

Figure 18: Screenshot of the input view.

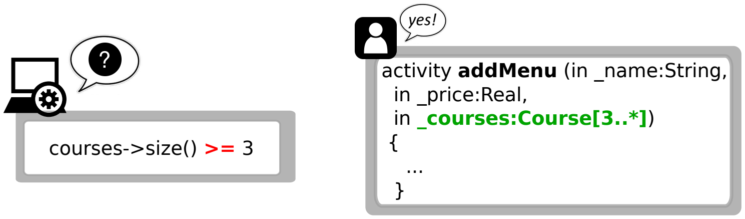
effect of a PVA could not be used and thus falsify the results of the method. Roughly, this SAT-problem could be tackled with UML/OCL verification tools [13] adding the test condition as an additional constraint to the model and checking if the extended model is still satisfiable. However, this analysis, would worsen the efficiency of our method.

Finally, we assume that operations to be analyzed do not include recursive invocations. This assumption is made because of recursive invocations generate infinite paths (given that the recursive invocation is replaced by the subdiagraph corresponding to the operation itself, then, this replacement process never finishes) that are not able to be addressed by our method. Nevertheless, recursive operations could be transformed into their iterative counterparts [6] before the application of our method.

Besides, our method presents several trade-offs that are required to enable our lightweight analysis.

As we introduced, one limitation of our method is the fact that our it performs an over-approximation analysis. This implies that it may classify as a ¬WE/¬SE an operation which is actually WE/SE (but not the other way round, the method never marks as WE/SE an operation which is not actually executable). This over-approximation can be resolved by the designer by participating in the second step of the method in order to disambiguate some situations that cannot be automatically computed without resorting to a search-based approach (which would then limit the benefits of the static analysis we perform). Our method is able to determine this in a number of cases but assumes the worst case scenario when it cannot be sure. Instead of extending the method with a simulation component to decide these situations (since, as said above, this would hinder the efficiency of the method), the designer could directly decide this by herself. The type of queries for which our method requires the user intervention are basic inequalities that a designer may easily solve by examining the operation.

Example 14 When verifying the operation addMenu the user intervention is required to determine whether “_courses→size() ≥ 3”. As Figure 19 shows, since the multiplicity of the _courses parameter in the addMenu operation is at least 3, the designer may easily conclude the above inequality is always true.



```
courses->size() >= 3
```

```
activity addMenu (in _name:String,
  in _price:Real,
  in _courses:Course[3..*])
 {
   ...
 }
```

Figure 19: Example of user intervention.

Our method covers the most commonly used integrity constraints. However, it is no suitable to address arbitrary integrity constraints, although some new constraints could be added to our patterns (adding the proper conditions in the tables of Steps 1 and 2 of the method).

7.2. Performance of our method

We consider our method is efficient (understanding the efficiency as the capability to perform the verification in a reasonable time) compared with other non-lightweight and formal methods that suffer from the state explosion problem (see some of them in Section 8).

Table 5 details the performance results for the experiments we conducted with our Eclipse plug-in in order to measure the running time when verifying the strong executability of several operations.

Note that the time complexity when verifying the weak executability is lower than the time complexity when verifying the strong executability, since in the former we must apply the method until we reach a WE path while in the second we must apply the method over all the paths of the operation.

This, together with the second main and distinguishing benefit of our method (the kind of feedback provided to the user), justify the above assumptions and limitations.

Table 5: Performance results for our prototype tool.

Operation	*Class Diagram size*	*Operation size*	*Running time*
`newCourse`	6 classes, 8 attributes, 4 associations, 1 generalization, 13 integrity constraints	3 actions, 1 loop	3641 ms
`addMenu`	6 classes, 8 attributes, 4 associations, 1 generalization, 13 integrity constraints	4 actions, 1 conditional, 1 loop	3579 ms
`classifyAsSpecialMenu`	6 classes, 8 attributes, 4 associations, 1 generalization, 13 integrity constraints	2 actions, 1 conditional	3522 ms
`scalabilityTest`	100 classes, 200 attributes, 40 associations, 10 generalizations, 100 integrity constraints	100 actions, 10 conditionals, 5 loops	4969 ms

8. Related Work

A lot of research has been devoted to the problem of V&V (verify and validate) UML models. In the context of UML behavioural models, there is a broad set of research proposals devoted to the analysis of statechart diagrams [29, 31, 42], sequence diagrams [24], activity diagrams [4, 7, 18, 30, 28, 33], operations [12, 22, 49], xUML models [25, 53], or on verifying the consistent interrelationship between them [9, 23], among others.

We classify the closest works wrt our method in Table 6 and position it in relation with them. For each approach (1st column) we indicate the kind of behavioural model targeted (2nd column), the integrity constraints that are supported when analyzing the models (3rd column), whether UML actions can be added to specify fine-grained details of the model (4th column), the main correctness properties addressed by the method (5th column), the employed method during the verification (6th column) and whether the approach returns some kind of feedback beyond a simple yes/no answer (7th column).

As Table 6 shows, half of the works do not support the definition of UML action sequences as part of the specification of their input behavioural models (even when this is indeed allowed by the UML standard). Analyze this detailed (i.e. fine-grained) specifications is precisely the focus of our method. Althought there are several works addressing the verification of UML models including actions [23, 25, 30, 53], only few works [4, 8, 16, 33, 28] are aligned with the new standard Alf action language.

On the other hand, works dealing with the executability of operations depart from declarative operations specified by means of pre and postconditions contracts, instead of using imperative operations. A comparison between declarative and imperative specifications is beyond the scope of this paper but it is worth to note that there are methods that transform declarative operations into imperative ones [13]. Once operations are transformed we can use the method presented herein to verify them in an efficient way.

Most of the above works simulate the behavioural models by translating them into Model Checking [5], Constraint Programming [26] or Query Containment [19], and thus, they present scalability issues. For instance, model checkers

Table 6: UML related methods comparison (shown chronologically).

Work	*Model*	*Supported Constraints*	*Include UML Actions?*	*Property*	*Method*	*Repairing Feedback?*
Paltor et al. [42]	Statechart diagram	No	No	Deadlocks, livelocks, etc.	Model checking	No
Latella et al. [29]	Statechart diagram	No	No	Safety, liveness	Model checking	No
Xie et al. [53]	xUML model	No	Yes (xUML)	Domain-specific properties	Model checking	No
Graw et at. [23]	Statechart diagram, sequence diagram	No	Yes (Action Semantics)	Consistency	Model checking	No
Grosu et al. [24]	Sequence diagram	No	No	Safety, liveness	Model checking	No
Eshius et al. [18]	Activity diagram	No	No	Safeness, etc.	Model checking	No
Bouabana-Tebibel et al. [7]	Activity diagram	No	No	Deadlocks, livelocks, liveness, etc.	Model checking	No
Gogolla et al. [22]	Declarative operations	Yes (all)	No	Validation checks	Animation	No
Cabot et al. [12]	Declarative operations	Yes (all)	No	WE, SE etc.	Constraint programming	No
Queralt et al. [49]	Declarative operations	Yes (subset)	No	WE etc.	Query containment	No
Hansen et al. [25]	xUML model	No	Yes (xUML)	Safety	Model checking	No
Brosch et al. [9]	Statechart diagram and sequence diagram	No	No	Consistency	Model Checking	No
Bousee et al. [8]	SysML statechart diagram	Yes	Yes (Alf)	Safety	Theorem proving	No
Lai et al. [28]	Activity diagram	No	Yes (Alf)	Basic redundancies	Static analysis	No
Abdelhalim et al. [4]	Activity diagram	No	Yes (fUML, Alf)	Deadlocks	Model checking	No
Craciun et al. [16]	Activity diagram	No	Yes (Alf)	Domain-specific properties	Testing	No
Laurent et al. [30]	Activity diagram	No	Yes (fUML)	Control-Flow, Data-Flow, Resources, Time, Business	Model Checking	No
Mijatov et al. [33]	Activity diagram	Yes (subset)	Yes (Alf)	Domain-specific properties	Testing	No
Our work	Imperative operations	Yes (subset)	Yes (Alf)	Weak and Strong Executability	Static analysis	Yes

work by generating and analyzing all the potential executions at run-time and evaluating if for each (or some) execution scenario the given property is satisfied. Even with the several optimizations available (as partial order reduction or state compression), methods based on model checking techniques suffer from the state-explosion problem (i.e. the number of potential executions to analyze grows exponentially) compromising the efficiency of the method. Instead, one of the main benefits or our method is its efficiency, understanding the efficiency as the capability to perform the verification in a reasonable time.

Finally, all the above methods just provide a binary response (if the model satisfies the given property or not) 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. Instead, another benefit of our method is the kind of feedback, that helps the designer repairing her models.

To sum up, our method is the only one that deals with the verification of UML operations including actions and provides repairing feedback.

9. Conclusions and Further Work

We have proposed a lightweight and static method for assisting designers during the specification of executable behavioural models. In particular, our method verifies the weak and the strong executability of action-based UML operations defined by means of the new standard Alf action language wrt the structural constraints imposed by the domain model.

The main benefits of our method are that it is lightweight (it directly reason over a model formalized in Alf language) and it is based on a static analysis of the model (no execution is required). This leads on a method that can be easily integrated in the current software development processes and CASE tools. But the more distinguishable benefit comparing with other methods is that our method provides a useful feedback to help the designers improve her models. Our method return either a positive answer (meaning that the model achieves the checked property) or a corrective feedback (otherwise) which is expressed in the same language used to express the input model.

As a trade-off, our method supports a limited (but still useful) set of integrity constraints and it may require the user intervention in order to return a more precise result. For these reasons, we believe the method presented in this paper could be used to perform a first correctness analysis, basic to ensure a fundamental quality level on action-based operations. Then, designers could proceed with a more detailed analysis adapting other methods (such as model checking) to perform a more complete verification (see Figure 20).

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

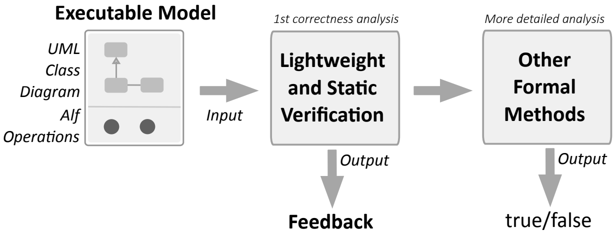


Figure 20: Connection with other verification methods.

References

[1] Alf-verifier: A lightweight tool for verifying UML-Alf executable models, https://github.com/som-research/alf-verifier.

[2] Alf-verifier: Example of use, https://goo.gl/pYEI9F.

[3] Xtext, www.xtext.org/ (Last visit September 2015).

[4] I. Abdelhalim, S. Schneider, and H. Treharne. An integrated framework for checking the behaviour of fUML models using CSP. *STTT*, 15(4):375–396, 2013.

[5] R. Alur. Model Checking: From Tools to Theory. In *25 Years of Model Checking*, pages 89–106, 2008.

[6] J. Arsac and Y. Kodratoff. Some Techniques for Recursion Removal from Recursive Functions. *ACM Trans. Program. Lang. Syst.*, 4(2):295–322, Apr. 1982.

[7] T. Bouabana-Tebibel and M. Belmesk. An Object-Oriented Approach to Formally Analyze the UML 2.0 Activity Partitions. *Information & Software Technology*, 49(9-10):999–1016, 2007.

[8] E. Bousse, D. Mentré, B. Combemale, B. Baudry, and K. Takaya. Aligning SysML with the B Method to Provide V&V for Systems Engineering. In *Model-Driven Engineering, Verification, and Validation 2012 (MoDeVVa 2012)*, Innsbruck, Autriche, Sept. 2012.

[9] P. Brosch, U. Egly, S. Gabmeyer, G. Kappel, M. Seidl, H. Tompits, M. Widl, and M. Wimmer. Towards Scenario-Based Testing of UML Diagrams. In *TAP*, pages 149–155, 2012.

[10] D. Brown. *An Introduction to Object-Oriented Analysis: Objects and UML in plain English.* Wiley, 2002.

[11] J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software*, 83(2):283–302, 2010.

[12] J. Cabot, R. Clarisó, and D. Riera. Verifying UML/OCL Operation Contracts. In *IFM*, volume 5423 of *LNCS*, pages 40–55, 2009.

[13] J. Cabot, R. Clarisó, and D. Riera. On the verification of UML/OCL class diagrams using constraint programming. *Journal of Systems and Software*, 93:1–23, 2014.

[14] C. Click and K. D. Cooper. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.*, 17(2):181–196, 1995.

[15] D. Costal, C. Gómez, A. Queralt, R. Raventós, and E. Teniente. Improving the definition of general constraints in UML. *Software and System Modeling*, 7(4):469–486, 2008.

[16] F. Craciun, S. Motogna, and I. Lazar. Towards Better Testing of fUML Models. In *ICST*, pages 485–486, 2013.

[17] Eclipse. Modeling Platform / Eclipse Con Europe Nov 2 2011. `http://wiki.eclipse.org/ModelingPlatform/EclipseConEuropeNov2_2011`, 2011.

[18] R. Eshuis. Symbolic Model Checking of UML Activity Diagrams. *ACM Trans. Softw. Eng. Methodol.*, 15(1):1–38, 2006.

[19] C. Farré, E. Teniente, and T. Urpí. Checking query containment with the CQC method. *Data Knowledge Engineering*, 53(2):163–223, 2005.

[20] V. Garousi, L. C. Briand, and Y. Labiche. Control flow analysis of UML 2.0 sequence diagrams. In *ECMDA-FA*, pages 160–174, 2005.

[21] M. Genero, M. Piattini, and M. R. V. Chaudron. Quality of UML models. *Information & Software Technology*, 51(12):1629–1630, 2009.

[22] M. Gogolla, F. Büttner, and M. Richters. USE: A UML-based specification environment for validating UML and OCL. *Sci. Comput. Program.*, 69(1-3):27–34, 2007.

[23] G. Graw and P. Herrmann. Transformation and Verification of Executable UML Models. *Electr. Notes Theor. Comput. Sci.*, 101:3–24, 2004.

[24] R. Grosu and S. A. Smolka. Safety-Liveness Semantics for UML 2.0 Sequence Diagrams. In *ACSD*, pages 6–14, 2005.

[25] H. H. Hansen, J. Ketema, B. Luttik, M. R. Mousavi, J. van de Pol, and O. M. dos Santos. Automated Verification of Executable UML Models. In *FMCO*, pages 225–250, 2010.

[26] 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.

[27] D. Harel. Biting the Silver Bullet - Toward a Brighter Future for System Development. *IEEE Computer*, 25(1):8–20, 1992.

[28] Q. Lai and A. Carpenter. Defining and verifying behaviour of domain specific language with fUML. In *Proceedings of the Fourth Workshop on Behaviour Modelling - Foundations and Applications*, BM-FA '12, pages 1:1–1:7, New York, NY, USA, 2012. ACM.

[29] 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.

[30] Y. Laurent, R. Bendraou, S. Baarir, and M. Gervais. Formalization of fUML: An Application to Process Verification. In *Advanced Information Systems Engineering - 26th International Conference, CAiSE 2014, Thessaloniki, Greece, June 16-20, 2014. Proceedings*, pages 347–363, 2014.

[31] J. Lilius and I. Paltor. vUML: A Tool for Verifying UML Models. In *ASE*, pages 255–258, 1999.

[32] S. J. Mellor. Executable UML Information Day - Keynote Presentation. `http://www.omg.org/news/meetings/tc/agendas/va/xUML_pdf/Mellor_Keynote.pdf(LastvisitSeptember2015)`, 2011.

[33] S. Mijatov, T. Mayerhofer, P. Langer, and G. Kappel. Testing functional requirements in UML activity diagrams. In *Tests and Proofs - 9th International Conference, TAP 2015, Held as Part of STAF 2015, L'Aquila, Italy, July 22-24, 2015. Proceedings*, pages 173–190, 2015.

[34] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[35] A. Nugroho and M. R. V. Chaudron. Evaluating the Impact of UML Modeling on Software Quality: An Industrial Case Study. In *MoDELS*, pages 181–195, 2009.

[36] A. Olivé. Conceptual Schema-Centric Development: A Grand Challenge for Information Systems Research. In *CAiSE*, pages 1–15, 2005.

[37] A. Olivé. *Conceptual Modeling of Information Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[38] OMG. UML 2.0 OCL Specification. (ptc/03-10-14). 2003.

[39] OMG. UML 2.4.1 Superstructure Specification. 2011.

[40] OMG. Concrete Syntax for UML Action Language (Action Language for Foundational UML), version 1.0.1, www.omg.org/spec/ALF (Last visit september 2015). 2013.

[41] OMG. Semantics Of A Foundational Subset For Executable UML Models (fUML), version 1.1, www.omg.org/spec/FUML/ (Last visit September 2015). 2013.

[42] I. Paltor and J. Lilius. Formalising UML State Machines for Model Checking. In *UML*, pages 430–445, 1999.

[43] I. Perseil. ALF formal. *ISSE*, 7(4):325–326, 2011.

[44] E. Planas. Lightweight and static verification of UML executable models. PhD Thesis. `http://www.tdx.cat/handle/10803/116449` (Last visit September 2015), 2013.

[45] E. Planas, J. Cabot, and C. Gómez. Verifying Action Semantics Specifications in UML Behavioral Models. In *CAiSE*, volume 5565 of *LNCS*, pages 125–140. Springer, 2009.

[46] E. Planas, J. Cabot, and C. Gómez. Lightweight Verification of Executable Models. In *ER*, volume 6998 of *LNCS*, pages 467–475. Springer, 2011.

[47] E. Planas, J. Cabot, and C. Gómez. Two Basic Correctness Properties for ATL Transformations: Executability and Coverage. In *MtATL*, pages 1–9, 2011.

[48] B. Pugh and A. Loskutov. FindBugs: A Static Analyzer Tool for Java Code, http://findbugs.sourceforge.net.

[49] A. Queralt and E. Teniente. Reasoning on UML Conceptual Schemas with Operations. In *CAiSE*, volume 5565 of *LNCS*, pages 47–62, 2009.

[50] H. Saiedian. An Invitation to Formal Methods. *Computer*, 29(4):16–17, Apr. 1996.

[51] M. J. B. Stephen J. Mellor. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley, 2002.

[52] UML2Tools. *http://www.eclipse.org/modeling/mdt/?project=uml2tools (Last visit September 2015)* .

[53] F. Xie, V. Levin, and J. C. Browne. Model Checking for an Executable Subset of UML. In *ASE*, pages 333–336, 2001.