c

A TOOL BASED ON DL FOR UML MODEL CONSISTENCY CHECKING

JOCELYN SIMMONDS

Computer Science Department, University of Toronto,
Room 3302, Sandford Fleming Building, 10 King's College Road,
Toronto, Ontario M5S 3G4, Canada
jsimmond@cs.toronto.edu

MARÍA CECILIA BASTARRICA*, NANCY HITSCHFELD-KAHLER[†]
and SEBASTIÁN RIVAS[‡]

Computer Science Department, Universidad de Chile,
Blanco Encalada 2120, Santiago, Chile
* *cecilia@dcc.uchile.cl*
[†] *nancy@dcc.uchile.cl*
[‡] *srivas@dcc.uchile.cl*

Automated consistency checking of UML models becomes necessary as models grow in size and complexity. Since the UML metamodel does not enforce model consistency, there are no fixed guidelines on how to approach the consistency problem. Current solutions are generally partial. The translation of the metamodel and the user designed model into Description Logics has proved to provide a solution in detecting a large set of inconsistencies. In order to make this solution available to system designers, we have implemented MCC+, a UML model consistency checker, built as a plug-in for Poseidon for UML, and relying on Jena as a reasoning engine. Compared to other approaches, we propose a usable and scalable solution, interoperable with a known modeling tool. We show the application of MCC+ to a real world large example of a meshing tool.

Keywords: Model consistency checking; UML modeling; description logics.

1. Introduction

UML, as a visual modeling language, provides a family of diagrams with which aspects like the structure and behavior of a system can be defined. A system design is defined by a model composed of a collection of diagrams where each one shows a different view of the system [7]. As these views are different representations of possibly overlapping definitions of the same system, inconsistencies could arise. The final system modeled using UML is a unique piece of software so inconsistencies in the design mean contradictory structure or behavior descriptions; this situation may mislead development teams or it may be impossible to use the model as a basis for automatic code generation.

The UML language is defined by its metamodel where only the elements that may be included in each diagram and the well formedness rules are established. The UML metamodel specification does not require that UML CASE tools implement consistency control either between model diagrams or among individual diagram elements. The main reason for this is that temporary model inconsistencies are not only introduced accidentally but sometimes are intentional, mainly when they are the result of intermediate steps during the design process. Transient inconsistencies can be tolerated, but final models must be consistent. As models grow larger and more complex, automatic consistency checking becomes necessary in order to determine whether a proposed model is valid or not.

Different approaches have been used to deal with inconsistencies in UML models, most of them treating specific types of inconsistencies with a particular formalism. There are also a few tools integrated into modeling frameworks for consistency checking. Theoretical feasibility is not enough for a formalism to be useful; it needs to be available for the user in the form of a user-friendly tool. We originally developed MCC [27], a framework that provided UML 2.0 model checking using automated reasoning Racer as the reasoning engine. Here we present the newer version, MCC+, where we substituted Racer for Jena, an open source DL based reasoning engine that improves the tool availability and avoids the need for tool configuration.

MCC+ uses knowledge representation systems with Description Logics (DL) [1] as a representation language. DL are decidable fragments of first-order logic that possess sound and complete reasoning mechanisms. In order to be able to reason about UML models using DL, the UML metamodel represents the domain knowledge and user models are translated as individual knowledge. These representations can be used for model consistency checking [31, 32]. A set of 18 different consistency checks for UML 1.5 were identified and the feasibility of solving them using DL was shown [26]. Although there are some scalability problems found in reasoning on UML class diagrams using DL, the complete size of the metamodel is fixed and bounded. Also, in our tool approach, we decided to calculate all the implicit knowledge when the user model is translated into DL so that querying the knowledge base is faster.

MCC+ is built as a plug-in integrated into Poseidon for UML, as a way of offering user-friendly consistency checking through a known user interface. We take advantage of the rich user interface provided by Poseidon, while using the power of the DL engine Jena [24] behind the scenes. Both Poseidon and Jena are robust systems, integrated using well defined public APIs.

We have designed a 3D moving boundary meshing software intended to model tree growth using UML diagrams. The model for this software manages around 30 classes, but it can manage thousands and even millions of instances at runtime. Checking for consistency in this model has a high impact on the quality of the resulting software, and also the inherent complexity of solving meshing problems could be faced relying on MCC+ for consistency details. We have applied the *abstract object*, *disconnected sequence diagram*, *disconnected state diagram* and *incompatible behavior* consistency checks in the application example. Even though theoretically DL could be a non-scalable solution, the strategy used in implementing our tool proved to be usable in this large system.

This paper is structured as follows. Related work is presented in Sec. 2. Section 3 explains how DL is used to reason about UML models and to check for consistency. The MCC+ tool is presented in Sec. 4. The application example is shown in Sec. 5 along with a discussion about complexity and scalability. Finally, some conclusions are included in Sec. 6.

2. Related Work

A wide range of different approaches for checking consistency in UML models has been proposed in the literature. Engels *et al.* [11] motivated a general methodology to deal with consistency problems based on the problem of protocol statechart inheritance. Communicating Sequential Processes (CSP) are used as a mathematical model for describing the consistency requirements. Ehrig and Tsiolakis [9] investigated the consistency between UML class and sequence diagrams. UML class diagrams are represented by attributed type graphs with graphical constraints, and UML sequence diagrams by attributed graph grammars. The problem of verifying whether the interactions expressed by a collaboration diagram can be realized by a set of state machines, has been treated by Schäfer *et al.* [25]. They have developed HUGO, a prototype tool that checks if state machines compiled into PROMELA [16] models, and collaborations translated into sets of Buchic automata match up, using the SPIN model checker to verify the model against the automata. This problem has also been analyzed by Litvak *et al.* [20], using an algorithmic approach, instead of using external model checkers. They have put their ideas into practice by implementing the BVUML tool, that receives the state and sequence diagrams as XMI files produced by ArgoUML. Using HUGO requires that the user translate his UML model into two different formalisms. This can be cumbersome and error-prone if the state machine that needs to be checked is complex, even though BVUML skips the user translation problem. MCC+ has a homogeneous approach for dealing with all consistency problems, using the metamodel definition as a basis for finding inconsistencies in user defined models.

Rational Rose [18], a popular UML CASE tool, also incorporates ad-hoc model consistency checking. It checks the whole model, applying all the checks available. The user cannot specify which consistency checks are applied. If the model is more or less complex, the process can be quite lengthy. The output that is presented to the user is cryptic and is displayed into the error log. The Rose Model Checker [21] is a script that takes the output from the model checks and gives it a more user-friendly interface, but it basically reports only broken links between model elements and model statistics, and does not provide more complex checks like incompatible behavior. MCC+ addresses model consistency checking on a one by one basis for several reasons: it is clearer for the user to select a type of consistency check based on the context and the diagram he/she is working on, and accordingly it is also easier to interpret the results. Also, performing a single check is certainly faster than checking the whole model and generally this all that is needed.

In Van Der Straeten *et al.* [30], DL is used as the underlying formalism, but Loom is used as the reasoning engine. MCC+ uses Jena, which is a more modern and flexible tool. Initially, in order to prove that DL was indeed of practical use in this domain, a rudimentary tool chain that offered some automation was devised [26]. This tool chain was successfully applied [28]. Van Der Straeten included these ideas in creating RACOoN [29] — which offers model refactorings in addition to that provided by MCC. RACOoN is based on UML 1.5 diagrams and presents a chain structure instead of the plug-in structure of MCC+ that manages UML 2.0 diagrams. Even though a tool chain allows interchangeability of the integrated tools more easily, it is less transparent for end users and its architecture does not promote high performance, a key issue in reasoning about UML diagrams [4].

The first MCC [27] used Racer [15] as its reasoning engine. Even though Racer is a robust and efficient product, it has become commercial in the last year, limiting MCC's availability. In MCC+ the reasoning engine has been substituted by Jena [24] because it follows the W3C standards OWL and RDF for resource description, and SPARQL for querying the knowledge base.

3. Reasoning about UML models using DL

UML is defined by its metamodel, and the metamodel only establishes the elements that may be included in each diagram as well as the well formedness rules. So, as UML lacks formal semantics, consistency problems may arise in models and diagrams. In order to allow manipulation of models and diagrams based on the meaning of the diagrams and not just their visual representation, a formalism is required.

Knowledge representation systems (KRS) are focused on providing high-level descriptions of problem domains, in order to reason and to allow the discovery of implicit consequences of the explicitly represented knowledge. We decided to use KRSs that have DL variants as the concept representation language for UML models for several reasons. The first one is that DL is decidable, that is, given a concept definition, it is possible to determine if this definition is consistent or not with the existing concept definitions. Given an instance definition, it can also be decided which is the concept definition that suits it most. DL also offers concept subsumption, that is, it builds a concept hierarchy by classifying their definitions, finding which are the more general concepts for a specific concept; we use subsumption when modeling generalizations between metaclasses. Finally, DL supports open-world semantics; this means that when translating UML models, considering that these models are rarely syntactically complete, we are able to specify incomplete instances, inferring the rest of the instance specifications from the concept definitions.

Even though reasoning on UML class diagrams using DL is EXPTIME-hard [4], the complete size of the metamodel is bounded to around 250 classes and each consistency check is independently applied, so the complexity of the reasoning only depends on the number of elements defined in the user defined model at hand. Moreover, MCC+ deals with only a part of the UML metamodel — class, component, state and sequence diagrams — so a universe of approximately 100 metamodel classes is considered. For the consistency checks already implemented, MCC+ currently deals with around 30 classes.

Modern DL reasoning engines are quite efficient, using tableau-based algorithms [13]. This is a key point when arguing about tool usability, as results of model checking should be available in a reasonable amount of time. Various reasoning engines based on description logics have been implemented, each with its own expressive power [14, 17, 19, 24]. We can take advantage of these tools and integrate them with existing UML CASE tools in order to be able to reason automatically about UML models. Each of these systems has a concept specification language that allows the definition of the terminology to be used in the creation of knowledge bases, where inferences can be performed later. The set of concept definitions is called the Terminological-Box (*Tbox*). The part of the knowledge base that contains the individuals that instantiate the concepts defined in the *Tbox*, is called the Assertional-Box (*Abox*). The *Abox* contains extensional knowledge about the domain of interest as a finite set of expressions relating concepts and relationships to individuals.

Figure 1 illustrates the relationship that exists between the UML metamodel and different user model diagrams. The metamodel shown is just a small part of the complete UML metamodel; we have included some of the metaclasses and meta-associations necessary for the specification of class, component and sequence diagrams. The metamodel is completely specified using class diagrams, and each element created in a user model instantiates the corresponding metaclass. For example, in the class diagram, `ShoppingCart` is a UML class, instantiating the UML metaclass `Class`. The association between `ShoppingCart` and `Customer` is an instance of the `Association` metaclass. The generalization relationship between `Customer` and `PremiumCustomer` instantiates the metaclass `Generalization`. These `<<instantiate>>` relationships also apply to the elements used in other diagrams. For example, in the component diagram, the component `Order` is an instance of the `Component` metaclass. In the sequence diagram, the objects instantiate the `Object` metaclass.

Figure 2 shows the translation into DL of a small part of the UML metamodel in Fig. 1, so it is part of the *Tbox*. `ModelElement`, the metaclass from which all metaclasses inherit, defines an attribute `name`. `Model` is by definition a `Model-Element`, as it inherits from this class, but it also defines the role `owned-element`, as a model is a namespace. In the same manner, `Class` is a `ModelElement` and defines two

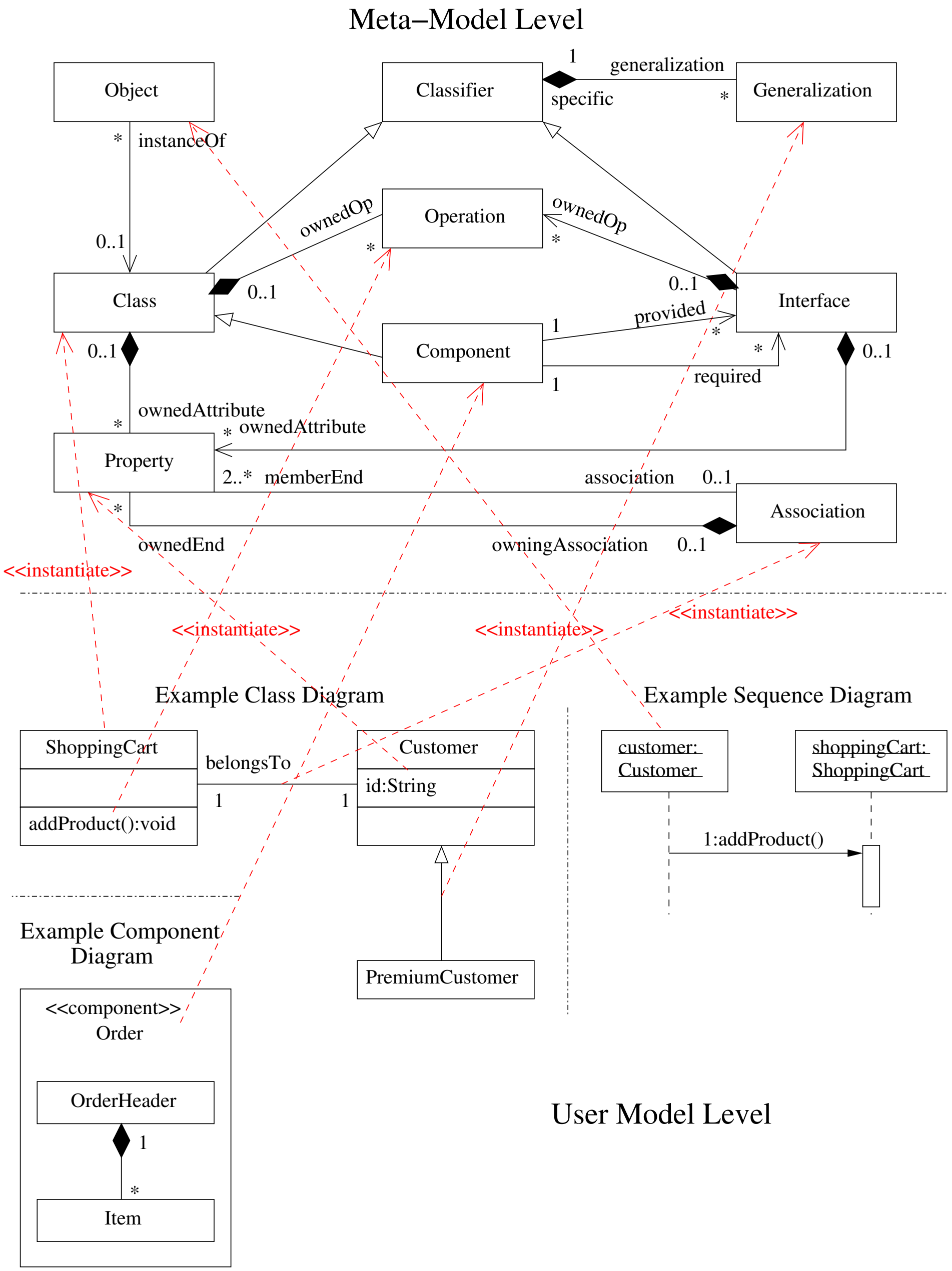


Fig. 1. Relationship between the UML metamodel and user models.

attributes: `isAbstract` and `isLeaf`. In the definition of the `Object` concept, qualified role restrictions are used to enforce the fact that an `Object` can only be the instance one `Class`.

```
(implies ModelElement (a name))
(implies Model
    (and ModelElement (all owned-element ModelElement)))

(implies Class
    (and ModelElement
    (a isAbstract)
    (a isLeaf)))

(implies Object
    (and ModelElement
    (exactly 1 instance-of)
    (all instance-of Class)))
```

Fig. 2. Part of the *Tbox* for Fig. 1.

Figure 3 shows a class diagram for an ATM machine model taken from [26] and originally developed by Russell Bjork for a computer science course at Gordon University [5]. Figure 4 shows the translation into DL of a part of this class diagram, and so it is part of the *Abox*. All the diagrams belong to a model called `model1`, so an instance of this concept is created and its name is set to the value “model1”.

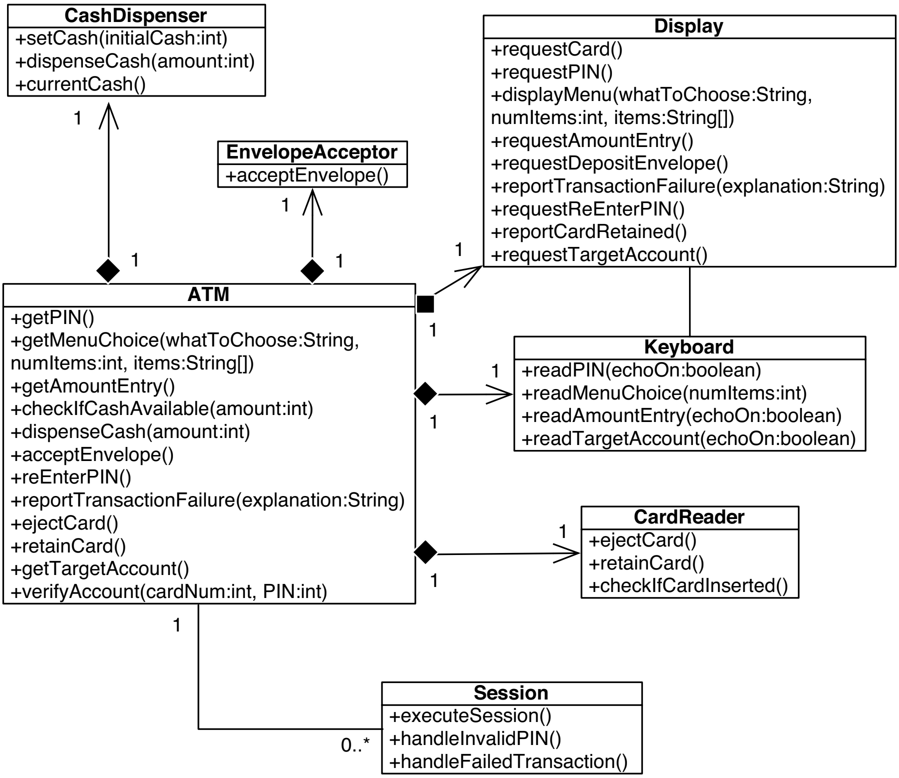

CashDispenser
+setCash(initialCash:int)
+dispenseCash(amount:int)
+currentCash()
EnvelopeAcceptor
+acceptEnvelope()
Display
+requestCard()
+requestPIN()
+displayMenu(whatToChoose:String, numItems:int, items:String[])
+requestAmountEntry()
+requestDepositEnvelope()
+reportTransactionFailure(explanation:String)
+requestReEnterPIN()
+reportCardRetained()
+requestTargetAccount()
ATM
+getPIN()
+getMenuChoice(whatToChoose:String, numItems:int, items:String[])
+getAmountEntry()
+checkIfCashAvailable(amount:int)
+dispenseCash(amount:int)
+acceptEnvelope()
+reEnterPIN()
+reportTransactionFailure(explanation:String)
+ejectCard()
+retainCard()
+getTargetAccount()
+verifyAccount(cardNum:int, PIN:int)
Keyboard
+readPIN(echoOn:boolean)
+readMenuChoice(numItems:int)
+readAmountEntry(echoOn:boolean)
+readTargetAccount(echoOn:boolean)
CardReader
+ejectCard()
+retainCard()
+checkIfCardInserted()
Session
+executeSession()
+handleInvalidPIN()
+handleFailedTransaction()
1
1
1
1
1
1
1
1
1
1
1
0..*

Fig. 3. Class diagram for an ATM machine model.

```
; Instance representing model1
(instance inst-model1 model)
(constrained inst-model1 name-of-model1 name)
(constraints (string= name-of-model1 “model1”))

; Instance representing ATM class
(instance inst-ATM class);
(constrained inst-ATM name-of-ATM name)
(constraints (string= name-of-ATM “ATM”))
(related inst-model1 inst-ATM owned-element)
(constrained inst-ATM abstract-ATM isAbstract)
(constraints (string= abstract-ATM “false”))
(constrained inst-ATM leaf-ATM isLeaf)
(constraints (string= leaf-ATM “true”))

; Instance representing anATM object
(instance inst-anATM object)
(constrained inst-anATM name-of-anATM name)
(constraints (string= name-of-anATM “anATM”))
(related inst-model1 inst-anATM owned-element)
(related inst-anATM inst-ATM instance-of)
```

Fig. 4. Part of the *Abox* (for Fig. 3).

The second set of definitions corresponds to the translation of the class `ATM`. First, its name is set to the value “ATM”. It is then related to the model instance `inst-model1`, as the `ATM` class belongs to `model1`. Extra details about the `ATM` class are stored: in this case the fact that the class is concrete and that it is a leaf. The last set of definitions corresponds to the translation of the object `anATM`. The name of the object instance is set to the value “anATM”. This object also belongs to `model1`, so it is also related to `inst-model1` using the `owned-element` role. Finally, the relationship between `ATM` and `anATM` is registered, relating the instances of the two model elements using the `instance-of` role.

The UML metamodel is completely defined in terms of class diagrams [22], and according to Cali *et al.* [6], class diagrams can be completely described in terms of DL, so the UML metamodel can be completely described in terms of DL. Also, all well formed user-defined UML models are instantiations of the UML metamodel [23], so all user defined UML models can also be completely described in terms of DL. In summary, both the *Tbox* and the *Abox* can be built using the information contained in the UML metamodel and the user defined model, respectively. Consistency checks are then implemented as queries to retrieve objects that obey certain conditions.

4. The Tool: MCC+

MCC+ (Model Consistency Checker) is an interactive tool that allows consistency checking at the UML level. This has the advantage that the system abstractions can be captured and shown in a visual manner. As a result, the authors believe that by giving the designer the opportunity to deal only with UML diagrams will make it easier to maintain evolving systems, as designers only focus on the modeling level of abstraction.

4.1. *Existing functionality*

A set of 18 consistency relationships not forced by the UML metamodel definition has been identified [26]. From this list, four consistency checks are already implemented as part of MCC+: abstract object, disconnected sequence diagram, disconnected state diagram, and incompatible behavior. Abstract object refers to the case in which an abstract class that has no concrete subclasses in the class diagrams of the model is instantiated in a sequence diagram. Disconnected sequence diagram arises when a sequence diagram has one or more objects that are not connected to the main diagram; it usually occurs when a lifeline is accidentally deleted. Disconnected state diagram ocurrs when a diagram has one or more states or transitions that are not connected with the main diagram; it usually happens when a state or a transition are accidentally deleted. Incompatible behavior arises when the ordered collection of stimuli received by an object in a sequence diagram does not exist as a sequence of events in the protocol state machine of the object's class.

The definition of these consistency checks was originally for UML 1.5 so they had to be updated to comply with the new UML specification (2.0). Most changes had to be applied to the predicates that involve sequence diagrams, as the metamodel for this diagram changed dramatically.

4.2. *Design decisions*

The Poseidon plug-in API is used to obtain the objects that represent the user model elements. These are passed to the translator, a singleton instance that returns the translation of the objects into DL. The translator instance uses the model element's dynamic type in order to determine which individual translation method should be invoked. This allows the seamless integration of translation methods for new element types and the application of changes to existing translations.

Model loading from the modeling tool into the inference engine and model checking by querying the knowledge base are independent actions. Any number of consistency checks can be applied without the extra overhead of re-translating the model between checks. The user only needs to reload the model when major changes are introduced.

In order to ensure usability, inconsistency detection and solution is a user activated process. This is due to the fact that while a model is being edited, it is usually in a temporary inconsistent state. Activating inconsistency detection automatically would imply developing predicate application strategies, for example, defining milestones at which model consistency should be checked, or macro-changes after which consistency should be checked. The development of this type of heuristics is an open problem [12]. Our decision has two direct consequences: the user is always in control and the implementation of the tool is simpler.

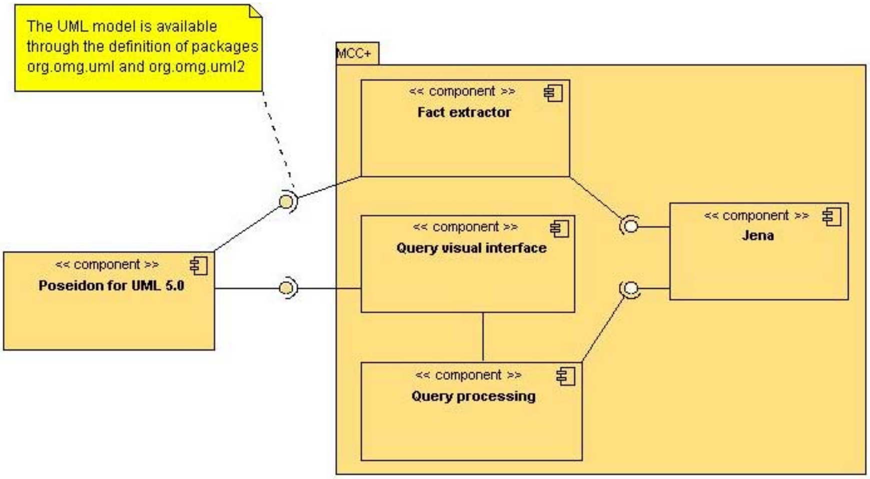


Fig. 5. MCC+ components.

4.3. *Implementation issues*

Figure 5 shows the components that make up the MCC+ tool and the relationships between MCC+, Poseidon and Jena.

- Query Visual Interface: interface that provides easy access to existing inconsistency detection predicates in an encapsulated manner. Also allows *Tbox* loading.
- Fact Extractor: provides facts needed in order to populate the *Abox* using a user-created UML model.
- Query Processing: acts as the communication channel between the Query Visual Interface and Jena.

Poseidon provides a graphical interface where UML models can be designed by dragging and dropping UML element templates. This software has a large amount of users because, even though it is not open-source, it is free (Community edition) and evaluation (Standard edition) versions are available.[a] It is implemented in Java, which makes it completely portable. This software is robust, currently version 4.2, and releases updates on a monthly basis. It also provides facilities for third-party extensions, through a plug-in API which allows access to all model elements and the graphical display. This makes MCC+ user-friendly for previous Poseidon users, as we only require that the user know UML and invoke the consistency checker through the existing, known interface.

Jena was chosen as the inference engine because it uses W3C standard languages, such as OWL-DL for ontology definition and SPARQL for queries. It provides a series of classes and methods that allow easy model manipulation, managing classes

[a] The plug-in requires the Standard edition, since plug-ins cannot be loaded into the Community version.

and instances, creating constraints, applying queries, reading and writing OWL-DL files, and may offer other functionalities defined by its API.

Creating a model in Jena is done through the class `ModelFactory`. It provides access for creating different types of models such as RDF and OWL, among others. In our case, we use the `createOntologyModel` with the `OntModelSpec.OWL_DL_MEM` parameter that specifies that an OWL-DL model will be managed in main memory. This method returns an instance of the `Model` class representing our ontology. This class also contains a series of methods that allow us to create classes, properties, data types, and constrains.

As an example, Fig. 6 shows the creation of a fragment of the *Tbox* represented in Fig. 7 using Jena's API.

```
//First we create an OWL-DL ontology
OntModel umlModel = ModelFactory.createOntologyModel(OntModelSpec.OWL_DL_MEM);
//We define a string that will be the basis for all other elements
String base = ”http://mcc/”;

//Now we create the class ModelElement using a String
OntClass ModelElement = umlModel.createClass(base + ”ModelElement”);
//We create the attribute name and we define it as a String
DatatypeProperty atributoName= umlModel.createDatatypeProperty(base +”atributoName”);
atributoName.addRange(XSD.xstring);
//Finally we state that ModelElement has a name
umlModel.createCardinalityRestriction(ModelElement.getURI(), atributoName, 1);

//We create the ontologic class Object
OntClass Object = umlModel.createClass(base + ”Object”);
//We state that Object inherits the properties from ModelElement
Object.addSuperClass(ModelElement);
//Finally we create the constraints over instance_of
umlModel.createCardinalityRestriction(Object.getURI(),instance_of, 1);
umlModel.createAllValuesFromRestriction(Object.getURI(),instance_of, Class);
```

Fig. 6. Partial *Tbox* created using Jena's API.

```
<!– ModelElement concept definition –>
<rdf:Description rdf:about=”http://mcc/ModelElement”>
    <rdf:type rdf:resource=”http://www.w3.org/2002/07/owl#Class”/>
    <rdf:type rdf:resource=”http://www.w3.org/2002/07/owl#Restriction”/>
    <owl:onProperty rdf:resource=”http://mcc/atributoName”/>
    <owl:cardinality rdf:datatype=”http://www.w3.org/2001/XMLSchema#int”>1
    </owl:cardinality>
</rdf:Description>

<!– Object concept definition –>
<rdf:Description rdf:about=”http://mcc/Object”>
    <rdf:type rdf:resource=”http://www.w3.org/2002/07/owl#Class”/>
    <rdfs:subClassOf rdf:resource=”http://mcc/ModelElement”/>
    <rdf:type rdf:resource=”http://www.w3.org/2002/07/owl#Restriction”/>
    <owl:onProperty rdf:resource=”http://mcc/instance_of”/>
    <owl:cardinality rdf:datatype=”http://www.w3.org/2001/XMLSchema#int”¿1¡/owl:cardinality>
    <owl:allValuesFrom rdf:resource=”http://mcc/Class”/>
</rdf:Description>
```

Fig. 7. Partial *Tbox* in OWL-DL.

Similarly, Jena provides an API for creating model individuals that populate the *Abox*. For this purpose it is necessary to use the `OntClass` objects already built and ask it to create an individual with a specific name. Afterwards, we can add the values of these individuals' attributes and properties. Figure 8 shows a fragment of the Jena code used for creating the OWL-DL individual presented in Fig. 9.

```
OntClass Class, Model;
OntProperty owned_element;
...

Individual model1 = Model.createIndividual(”inst-model1”)
...

Individual client = Class. createIndividual(”inst-Client”);
client.addProperty(attribute_name, ”Client”);
client.addProperty(owned_element, model1);
client.addProperty(attributeIsAbstract, false);
client.addProperty(attributeIsLeaf, false);
```

Fig. 8. Creation of an individual of class `Client` using the Jena API.

```
<!– Individual representing the Client class –>
<rdf:Description rdf:about=”http://mcc/inst-Client”>
    <rdf:type rdf:resource=”http://mcc/Class”/>
    <j.0:attributeName>Client</j.0:attributeName>
    <j.0:owned_element rdf:resource=”http://mcc/inst-model1”/>
    <j.0: attributeIsAbstract>false</j.0: atributeIsAbstract>
    <j.0: attributeIsLeaf>false</j.0: attributeIsLeaf>
</rdf:Description>
```

Fig. 9. OWL-DL individual of the `Client` class.

Similar to the one for creating ontologies, Jena provides an API for creating and executing SPARQL queries. For this purpose it is necessary to use the `QueryFactory` for creating a query from a `String`, and `QueryExecutionFactory` for instantiating the execution of a query created over a certain model. Afterwards, and using the object returned by `QueryExecutionFactory`, the query can be executed and we obtain a result set that can be sequentially traversed. If we execute the code in Fig. 10 we will obtain the SPARQL query in Fig. 11, and executing the query on the model we will obtain the result "Name: Client".

5. Application Example

The development of meshing technologies has become an intense theoretical and practical research area. In spite of its complexity, and perhaps due to its complexity, only in the last years the development of meshing software has been researched from the software engineering point of view, mainly by using formal methods for improving reliability of mesh generation software [10], by applying object-oriented design and programming [2], and by proposing a meshing tool architecture for a family of meshing tool [3].

```
String NL = System.getProperty(”line.separator”);
String queryString = ”PREFIX j.0: <http://mcc/>” + NL +
             ” SELECT ?name ” +
             ” WHERE { ” +
             ” <http://mcc/ins-Client> j.0:attributeName?name” +
             ” } ”;
Query query = QueryFactory.create(queryString) ;
QueryExecution qexec = QueryExecutionFactory.create(query, umlModel) ;
try {
         ResultSet rs = qexec.execSelect() ;

         while(rs.hasNext()){
             QuerySolution rb = rs.nextSolution() ;
             Resource name = rb.getResource(”name”).getString();
             System.out.println(”Name: ” + name);
         }

} finally{
         qexec.close() ;
}
```

Fig. 10. Building a SPARQL query using Jena.

```
PREFIX j.0: <http://mcc/>

SELECT ?name
WHERE { <http://mcc/ins-Client> j.0:attributeName ?name }
```

Fig. 11. SPARQL query.

5.1. *The meshing tool*

One of the meshing tools that we have developed using object-oriented techniques is a tree growth mesh generator (TGMG). We here describe part of the TGMG UML model and how its design was checked using MCC+.

The auxin is a plant hormone that influences the regulation of the tree cell division and cell expansion. There is a mathematical model for the auxin transport in conifer trees including gravity dependence that was implemented using Comsol [8], and TGMG generates the appropriate meshes for this model.

The tree growth modeling process is as follows:

- Generate (or read) a proper initial surface mesh that represents the tree surface
- For each simulation step:
  - — Simulate the phenomena using Comsol in order to compute the offset of each mesh point. The offset is defined by a vector.
  - — Build the new surface mesh
    - * Compute the new coordinates of each mesh point
    - * Detect possible collisions
    - * Solve the collisions
  - — Improve/refine/derefine the mesh according to the modeling requirements

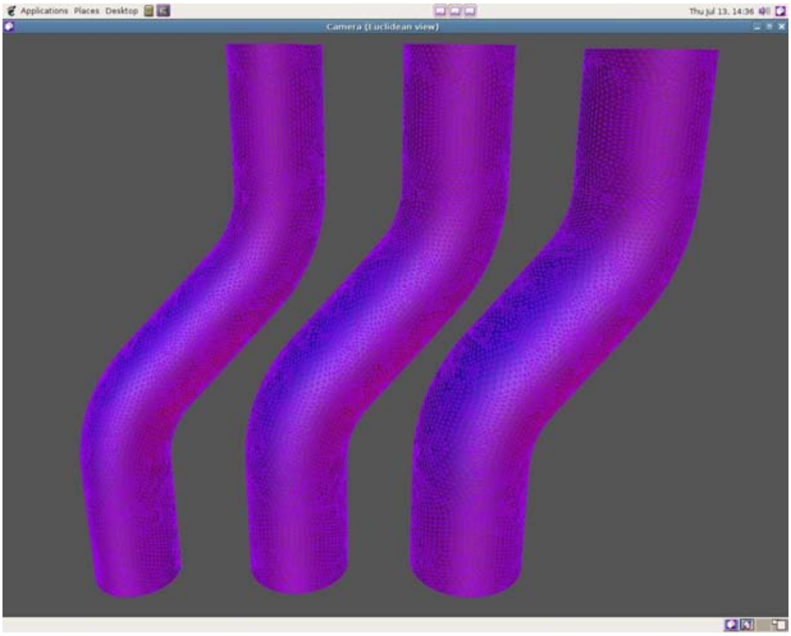


Fig. 12. Moving boundary meshing tool.

The tool's user interface lets the user choose from a list of possible moving boundary, improvement and refinenement strategies. Among the moving boundary strategies, we can find `LocalVerification` and `NoVerification`. Figure 12 shows an example of a sequence of tree stem meshes where the left one represents the initial situation and the following are two possible simulated growths.

For the tool both modeling elements and algorithms were designed as classes in order to achieve modifiability by extensibility. This diverse set of classes and their sophisticated interaction yield a complex model, possibly difficult to test. The tool class diagram has around 30 classes and it is shown in Fig. 13. Figure 14 shows the state diagram for the `Mesh` class; here the correct use of the mesh is described: first it needs to be initialized, then we can ask for its nodes, then query it for its edges and only then it is possible to check for its values of volume zero generated when two neighbor mesh elements overlap.

Finally, Fig. 15 shows a sequence diagram representing the `Move` operation using the `LocalVerification` algorithm. We first apply the initialization of the moving algorithm to be used which, in this case, is the `LocalVerification` algorithm. Next, by using this algorithm, all edges are traversed so that the first inconsistency can be detected, if it actually exists. Once a proportion value is determined, all vertices are traversed so that they are moved a distance proportional to this value. If there is no collision, all vertices are moved following the complete distance specified by the simulation step.

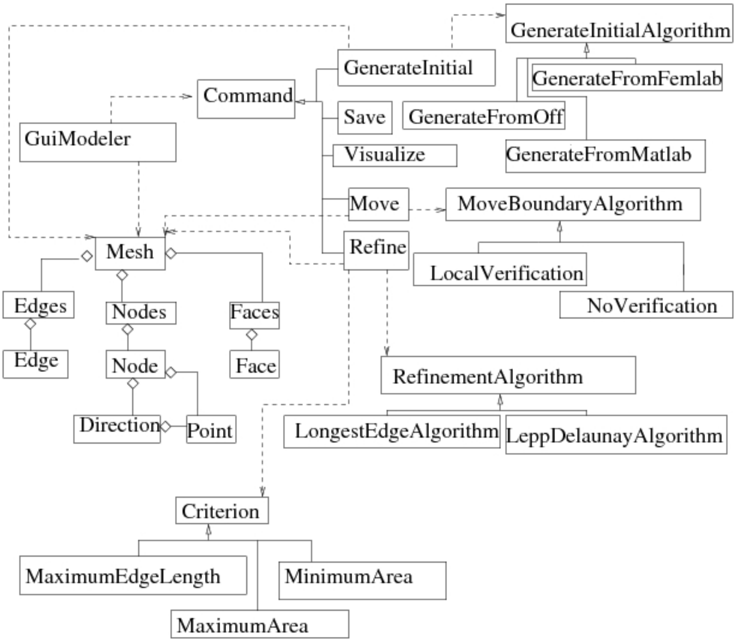


Fig. 13. Class model for the TGMG tool.

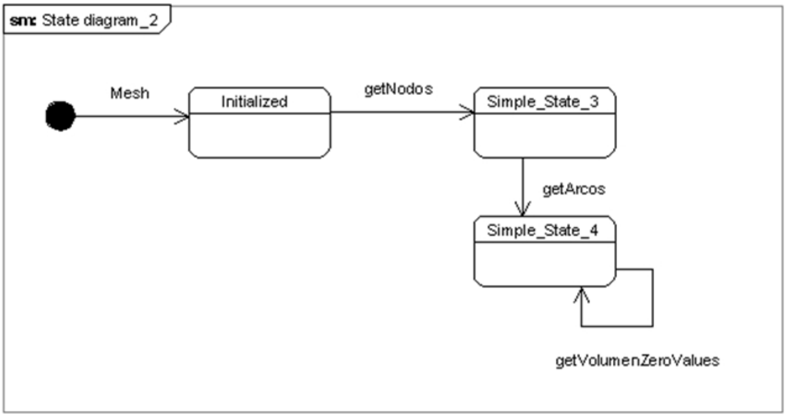


Fig. 14. State diagram for the Mesh class.

5.2. *Applying MCC+*

Using MCC+ requires three steps: loading the plug-in, translating the user defined model into DL assertions, and applying concrete consistency checks to the model. These three operations are applied at different times during the user work, and optimizing one of them may make the others worse. In the design of MCC+ we

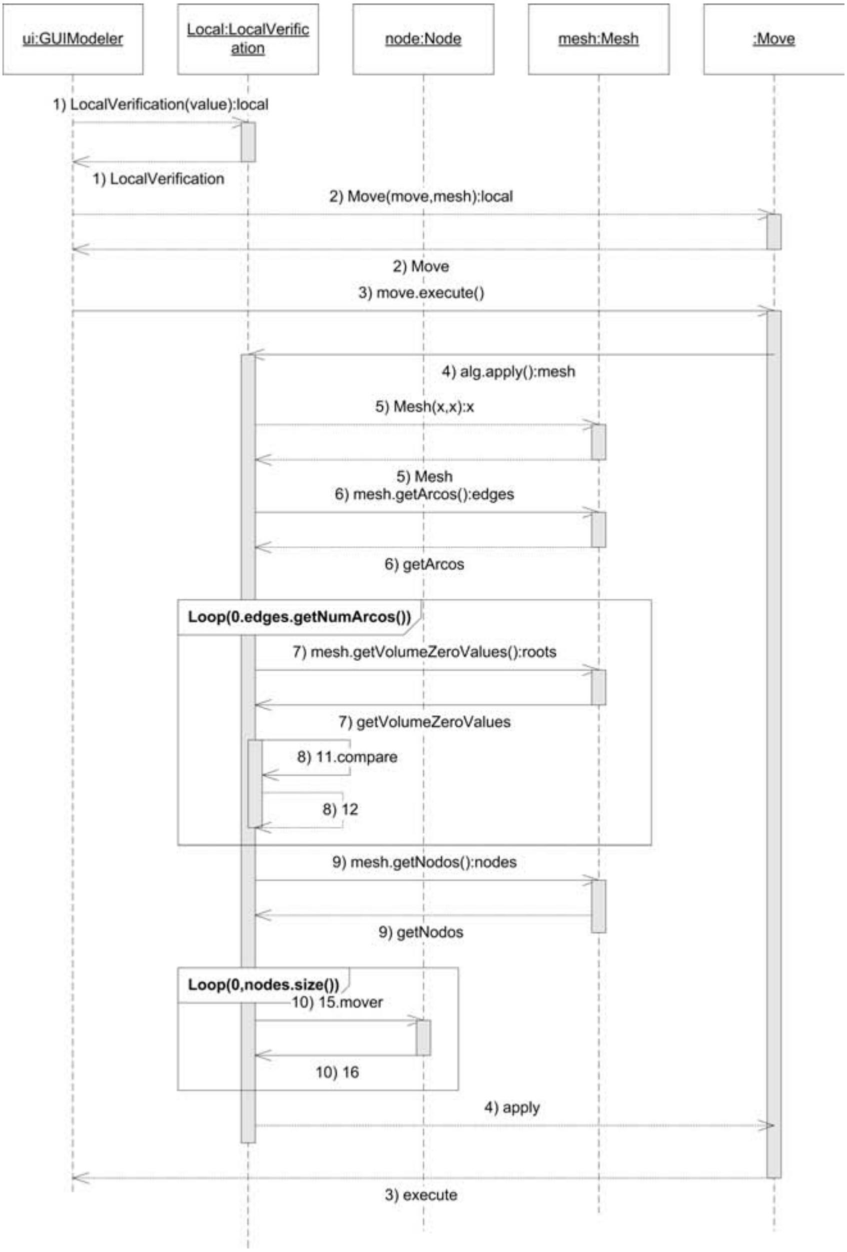


Fig. 15. Sequence diagram for the Move operation.

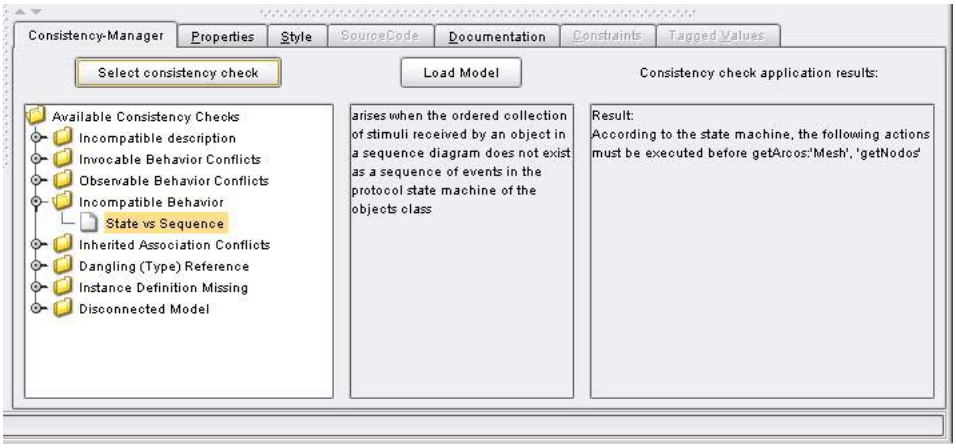
take the approach of including as much processing as possible before the model checking, so that this process that may be repeatedly applied can be faster.

When loading MCC+ as a plug-in, it translates the UML metamodel into DL and loads it as the *Tbox*; this operation is independent of the model at hand. The complete operation currently takes around ten seconds.

Once the model is complete or partially complete, or when the user decides to apply certain consistency checks, the model needs to be translated into DL in order to populate the *Abox*. In the case of the meshing tool, the whole assertion generation took around five seconds.

Loading the *Abox* not only inserts assertions into Jena, but also calculates all the implicit knowledge and stores it in the knowledge base. This extra work makes it possible to save processing time during model consistency checking. This design decision allows MCC+ to balance the work load, provided that multiple consistency checks may be applied before the model is changed and it should be reloaded. All this performance information certainly depends on the power of the computer at hand, but it provides a clear idea of a reasonable response time, mainly if compared to the large amount of time that may be saved during development due to fewer inconsistencies in the models.

The example in Figs. 13–15 represents only a small portion of the complete model for representing the application. The presence of inconsistencies in these diagrams may clearly result in several errors during the application development. All implemented consistency checks were applied to the example model. For the three basic checks, the results were successful. However, when the "incompatible behavior" was applied, the result was what is shown in Fig. 16.


Consistency-Manager
Properties
Style
SourceCode
Documentation
Constraints
Tagged Values
Select consistency check
Load Model
Consistency check application results:
Available Consistency Checks
Incompatible description
Invocable Behavior Conflicts
Observable Behavior Conflicts
Incompatible Behavior
State vs Sequence
Inherited Association Conflicts
Dangling (Type) Reference
Instance Definition Missing
Disconnected Model

arises when the ordered collection of stimuli received by an object in a sequence diagram does not exist as a sequence of events in the protocol state machine of the objects class

Result:
According to the state machine, the following actions must be executed before getArcos:'Mesh', 'getNodos'

Fig. 16. Applying "incompatible behavior" consistency check to Figs. 13, 14 and 15.

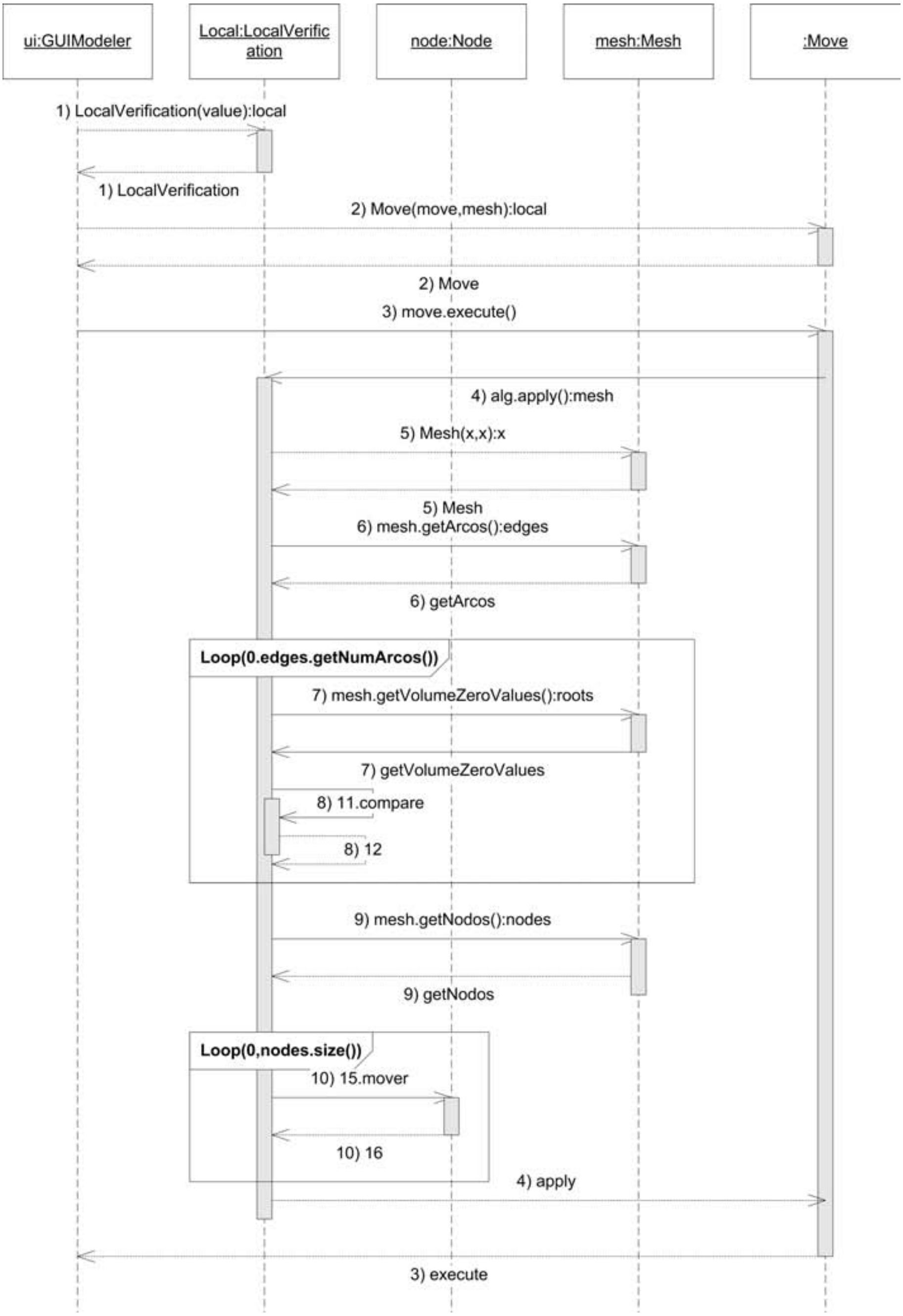
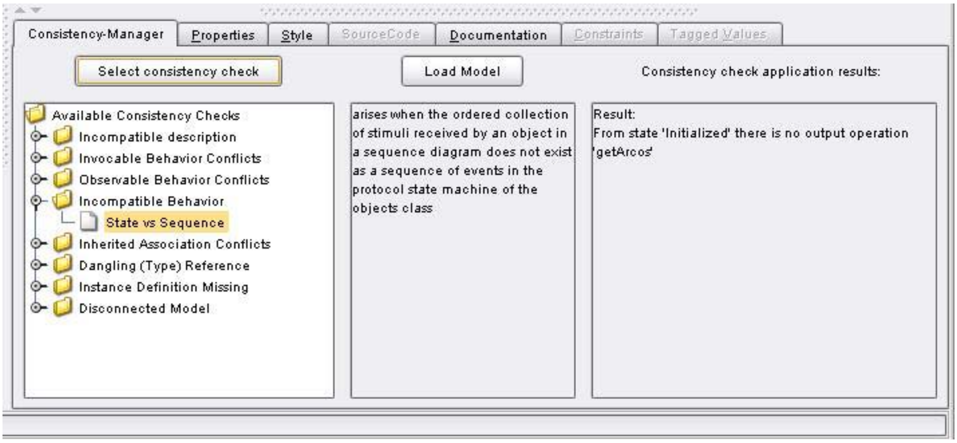


Fig. 17. Corrected sequence diagram for the Move operation.


Consistency-Manager
Properties
Style
SourceCode
Documentation
Constraints
Tagged Values
Select consistency check
Load Model
Consistency check application results:
Available Consistency Checks
Incompatible description
Invocable Behavior Conflicts
Observable Behavior Conflicts
Incompatible Behavior
State vs Sequence
Inherited Association Conflicts
Dangling (Type) Reference
Instance Definition Missing
Disconnected Model

arises when the ordered collection of stimuli received by an object in a sequence diagram does not exist as a sequence of events in the protocol state machine of the objects class

Result:
From state 'Initialized' there is no output operation 'getArcos'

Fig. 18. Applying "incompatible behavior" consistency check to Figs. 13, 14 and 17.

The observed results indicate that it is necessary to initialize the instance of the `Mesh` by executing the class constructor before invoking any other operation. The result of making this modification is shown in Fig. 17.

We reapply the "incompatible behavior" check and the results obtained are shown in Fig. 18. The errors reported reveal that according to the state diagram in Fig. 14, after instantiating the `Mesh` class, we get to the "Initialized" state and it does not have the "getArcos" operation as a valid output. Thus, it can be deduced by examining both diagrams, the problem is that we need to invoke the "getNodos" operation before invoquing "getArcos". Fixing this error in the sequence diagram, we get the one shown in Fig. 19. Rechecking for "incompatible behavior" we no longer have any errors of this type, as shown in Fig. 20.

6. Conclusion

We offer a generic tool for working with evolving UML models, with functional consistency checking schemes. Unlike other studies that are dedicated to the exhaustive study of a few consistency problems with a certain technique, we can detect a large set of possible consistency problems and we use the same approach to deal with them all. We believe that this approach will be considered useful by users, as it provides uniform support for the different consistency problems, with the possibility of integrating additional consistency detection easily. By using this tool, users can generate models that are consistent through a user-friendly tool.

The currently available version of MCC+ is an academic prototype, providing a reduced list of implemented consistency checks. The remaining identified checks must also be added to the tool. New checks concerning other diagrams can also be included, as well as translating new diagrams.

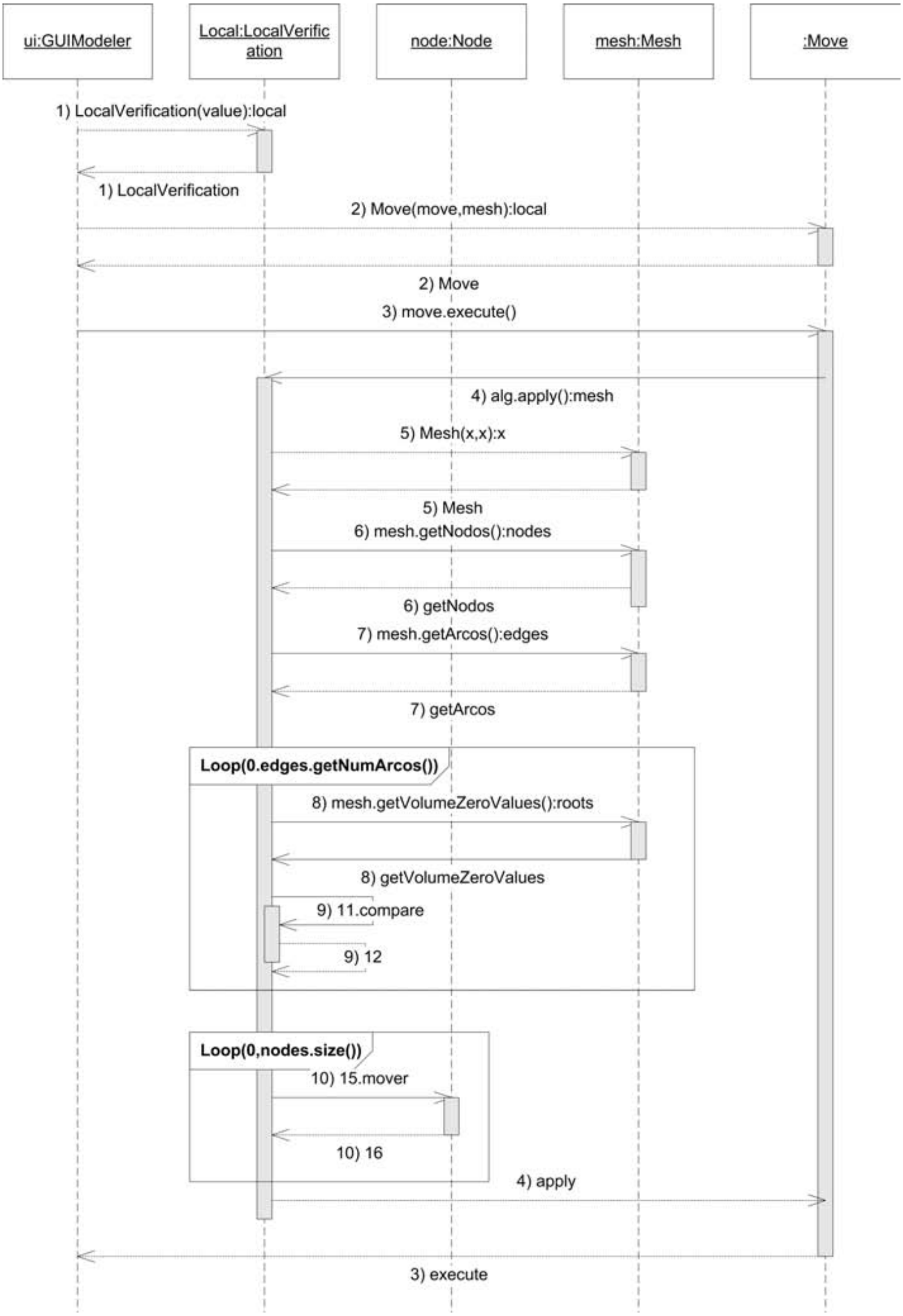
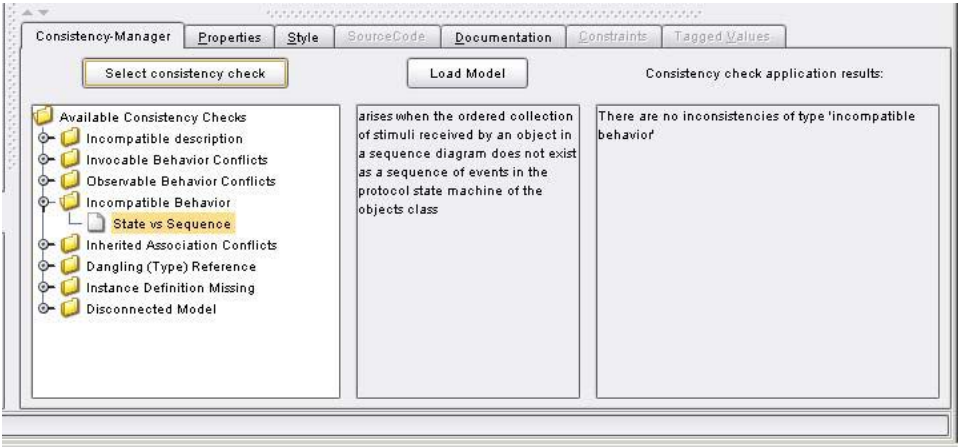


Fig. 19. Final sequence diagram for the Move operation.


Consistency-Manager
Properties
Style
SourceCode
Documentation
Constraints
Tagged Values
Select consistency check
Load Model
Consistency check application results:
Available Consistency Checks
Incompatible description
Invocable Behavior Conflicts
Observable Behavior Conflicts
Incompatible Behavior
State vs Sequence
Inherited Association Conflicts
Dangling (Type) Reference
Instance Definition Missing
Disconnected Model

arises when the ordered collection of stimuli received by an object in a sequence diagram does not exist as a sequence of events in the protocol state machine of the objects class

There are no inconsistencies of type 'incompatible behavior'

Fig. 20. Applying "incompatible behavior" consistency check to Figs. 13, 14 and 19.

In this paper we showed that MCC+ is not only able to check academic toy examples, but also real world real size complex examples in a reasonable time. Meshing tools are very good candidates for consistency checking because they are inherently complex pieces of software, naturally object-oriented and potentially composed of millions of objects at runtime.

Acknowledgments

The work of Nancy Hitschfeld-Kahler has been supported by project Fondecyt No. 1061227.

References

1. F. Baader, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, *The Description Logic Handbook: Theory, Implementation and Applications* (Cambridge University Press, 2003).
2. M. C. Bastarrica and N. Hitschfeld-Kahler, Designing a product family of meshing tools, *Advances in Engineering Software* **37**(1) (2006) 1–10.
3. M. C. Bastarrica, N. Hitschfeld-Kahler, and P. O. Rossel, Product line architecture for a family of meshing tools, in *Proc. 9th Int. Conf. Software Reuse* (*ICSR 2006*), LNCS 4039, Turin, Italy, June 2006.
4. D. Berardi, D. Calvanese, and G. De Giacomo, Reasoning on UML class diagrams is EXPTIME-hard, in *Proc. 2003 Description Logic Workshop* (*DL 2003*), CEUR Electronic Workshop Proceedings, 2003, pp. 28–37.
5. R. Bjork, Atm simulation links, 2000, http://www.math-cs.gordon.edu/courses/cs211/ATMExample/Links.html.
6. A. Calì, D. Calvanese, G. De Giacomo, and M. Lenzerini, A formal framework for reasoning on UML class diagrams, in *Proc. 13th Int. Symp. Methodologies for Intelligent Systems* (*ISMIS 2002*), LNCS Vol. 2366, Springer, 2002, pp. 503–513.
7. P. Clements, F. Bachman, I. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and

J. Stofferd, *Documenting Software Architectures. Views and Beyond* (Addison-Wesley, 2002).

8. Comsol, Comsol, 2006, http://www.comsol.com/.
9. H. Ehrig and A. Tsiolakis, Consistency analysis of UML class and sequence diagrams using attributed graph grammars, in *ETAPS 2000 Workshop on Graph Transformation Systems*, eds. H. Ehrig and G. Taentzer, March 2000, pp. 77–86.
10. A. H. ElSheikh, W. S. Smith, and S. E. Chidiac, Semi-formal design of reliable mesh generation systems, *Advances in Engineering Software* **35**(12) (2004) 827–841.
11. G. Engels, R. Heckel, and J. M. Küster, Rule-based specification of behavioral consistency based on the UML meta-model, in *Proc. Int. Conf. UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, eds. M. Gogolla and C. Kobryn, LNCS Vol. 2185, Springer-Verlag, 2001, pp. 272–286.
12. J. C. Grundy, J. G. Hosking, and W. B. Mugridge, Inconsistency management for multiple-view software development environments, *IEEE Trans. Software Engineering* **24**(11) (1998) 960–981.
13. V. Haarslev and R. Möller, High performance reasoning with very large knowledge bases: A practical case sttudy, in *Proc. 17th Int. Joint Conf. on Artificial Intelligence* (*IJCAI 2001*), 2001, pp. 161–168.
14. V. Haarslev and R. Möller, RACER, April 8 2003, http://www.fh-wedel.de/~mo/racer/.
15. V. Haarslev, R. Möller, R. Van Der Straeten, and M. Wessel, Extended query facilities for RACER and an application to software-engineering problems, in *Proc. Int. Workshop in Description Logics 2004* (*DL 2004*), 2004.
16. G. J. Holzmann, The model checker spin, *IEEE Trans. Software Engineering* **23**(5) (1997) 279–295.
17. I. Horrocks, FaCT, September 2004, http://www.cs.man.ac.uk/~horrocks/FaCT/.
18. IBM, Rational Software, October 2004, http://www-306.ibm.com/software/rational/.
19. ISI, Loom, April 2003, http://www.isi.edu/isd/LOOM/LOOM-HOME.html.
20. B. Litvak, S. Tyszberowicz, and A. Yehudai, Consistency validation of UML diagrams, in *Correctness of Model-Based Software Composition* (*CMC*) *Workshop* (*ECOOP*), 2003.
21. M. Moors, Rose Model Checker, October 2004, http://www.rationalrose.com/modelchecker/index.htm.
22. Object Management Group, UML 2.0 Infrastructure Specification, September 2003, http://www.omg.org/docs/ptc/03-09-15.pdf.
23. Object Management Group, UML 2.0 Superstructure Specification. 04-05-02.pdf, May 2004, http://www.omg.org/docs/ptc/.
24. HP Labs Semantic Web Programme, Jena — a semantic web framework for Java, November 2006, `http://jena.sourceforge.net/`.
25. T. Schäfer, A. Knapp, and S. Merz, Model checking UML state machines and collaborations, *Electronic Notes in Theoretical Computer Science* **47** (2001) 1–13.
26. J. Simmonds, *Consistency Maintenance of UML Models with Description Logics*, Master's thesis, Vrije Universiteit Brussel, Belgium and Ecole des Mines de Nantes, France, 2003.
27. J. Simmonds and M. C. Bastarrica, A tool for automatic UML model consistency checking, in *Proc. 20th IEEE/ACM Int. Conf. on Automated Software Engineering* (*ASE'2005*), Long Beach, CA, November 2005, pp. 431–432. Tool demonstration track.
28. R. Van Der Straeten, Inconsistency detection between UML models using RACER and nRQL, in *Proc. KI-2004 Workshop on Applications of Description Logics* (*ADL'04*),

University of Ulm, Germany, September 2004.

29. R. Van Der Straeten, *Inconsistency Management in Model-Driven Engineering*, PhD thesis, Vrije Universiteit Brussel, September 2005.
30. R. Van Der Straeten, T. Mens, and J. Simmonds, Maintaining consistency between UML models with description logic tools, in *ECOOP Workshop on Object-Oriented Reengineering*, Darmstadt, Germany, July 2003.
31. R. Van Der Straeten, T. Mens, J. Simmonds, and V. Jonckers, Using description logics to maintain consistency between UML models, in *Proc. Unified Modeling Language: Modeling Languages and Applications* (*UML 2003*), LNCS Vol. 2863, Springer-Verlag, Heidelberg, October 2003, pp. 326–340.
32. R. Van Der Straeten, J. Simmonds, and T. Mens, Detecting inconsistencies between UML models using description logic, in *Proc. 2003 Int. Workshop on Description Logics* (*DL 2003*), September 2003.