Description Logics for Consistency Checking of Architectural Features in UML 2.0 Models

Jocelyn Simmonds and M. Cecilia Bastarrica

Departamento de Ciencias de la Computación
Facultad de Ciencias Físicas y Matemáticas
Universidad de Chile
{jsimmond,cecilia}@dcc.uchile.cl

Abstract. UML has become the de facto standard language for software modeling. Although it was first created as a language for documenting detailed object-oriented designs, its newest version -UML 2.0- introduced new features for documenting software architectures as part of the standard. Models in UML include a series of diagrams that describe different views of the system, and even though the standard does not enforce consistency, good software engineering practices do. MCC is a tool based on description logics for UML model checking. In this paper we present how MCC can be used for reasoning about the consistency of UML models and how it can be extended so that it can also deal with UML 2.0 new architectural features following the same strategy.

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 diagram shows a different view of the system [4]. As these different views are different representations of possibly overlapping system definitions, inconsistencies could arise due to omissions or lack of standardization.

The standard for UML 2.0 is defined by its metamodel. This specification does not require that UML CASE tools implement consistency control neither between model diagrams nor among individual model elements. The main reason why this is so is because temporal model inconsistencies are not only introduced accidentally but sometimes also desirable, mainly when these are the result of intermediate steps during the design process. For example, there can be incomplete classifier specifications, or links that reference non-existing operations as the responsible for the operation still has not been determined. These inconsistencies can be tolerated during the design phase, but final models must be consistent. As models grow larger and more complex, automatic consistency checking between the different diagrams and model elements is necessary in order to determine whether a proposed model is valid or not.

One of the most salient new features introduced in the new UML 2.0 [22, 23] is the expressiveness for component diagrams, enhancing the usability of UML

as an Architecture Description Language (ADL) [20]. But having yet another specification dimension also brings new opportunities for inconsistencies.

Knowledge representation systems (KRS) are focused on providing high-level descriptions of problem domains, in order to allow the discovery of implicit consequences of the explicitly represented knowledge. Of special interest for our work are KRSs using Description Logics (DL) [2] as a representation language, as DL is a decidable fragment of first-order logic that possesses sound and complete reasoning mechanisms. In order to be able to reason about UML models, the UML metamodel is used to establish the domain representation and user models are translated as individual knowledge. In [3], it is proved that DL can be used to represent and reason about class diagrams. Translating both, the UML metamodel and user defined models into DL, we can reason about them and check for consistency.

We have built MCC[1], a framework that provides UML 2.0 model checking using automated reasoning provided by a DL implementation. It already includes six different consistency checks. This article presents a new feature of MCC for checking an architecture related inconsistency: service availability. Adding this capability, we also show that MCC is an extensible tool that can incorporate new checks with the same strategy followed in the previous ones. MCC is a plug-in integrated into an existing UML 2.0 CASE tool, 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 Racer 1.7 [11] behind the scenes. Both the UML CASE tool and the DL engine are robust systems, seamlessly integrated using well defined public APIs.

The paper is structured as follows. In Section 2, a description of the usage of DL as a means for reasoning about UML models is presented. In Section 3, the MCC tool is described. New architectural features included as part of the UML 2.0 standard are explained in Section 4, and it is also shown how MCC is extended in order to apply consistency checks on some of these architectural features of UML models. A discussion on related work is included in Section 5. Finally, some conclusions and a description of the ongoing work are presented in Section 6.

2 Reasoning about UML models using DL

UML is defined by its metamodel, and it 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. Thus, 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

[1] MCC can be found at `http://www.dcc.uchile.cl/~jsimmond`.

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 most suits it. 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, as these models are rarely syntactically complete, we are able to specify incomplete instances, inferring the rest of the instance specifications from the concept definitions.

Besides all the theoretical reasons, it is also important to consider that modern DL reasoning engines are quite efficient, using tableau-based algorithms. This is a key point when arguing tool usability, as results of model checking should be available in a reasonable amount of time.

Various systems [11, 14, 18] based on description logics have been implemented, each with its own expressive power. We can take advantage of these tools and integrate them with existing UML CASE tools in order to add formal semantics to 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 later be performed. 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. Greater detail about DL, *Tbox* and *Abox* can be found in [1].

Figure 1 shows the relationship that exists between the UML metamodel and different user models. The metamodel shown is just a small part; in this case, it includes 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. 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 `Generalization` metaclass.

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. The classes that implement the services offered by the component obey the same specification as classes that belong to a class diagram. In the sequence diagram, the objects instantiate the `Object` metaclass.

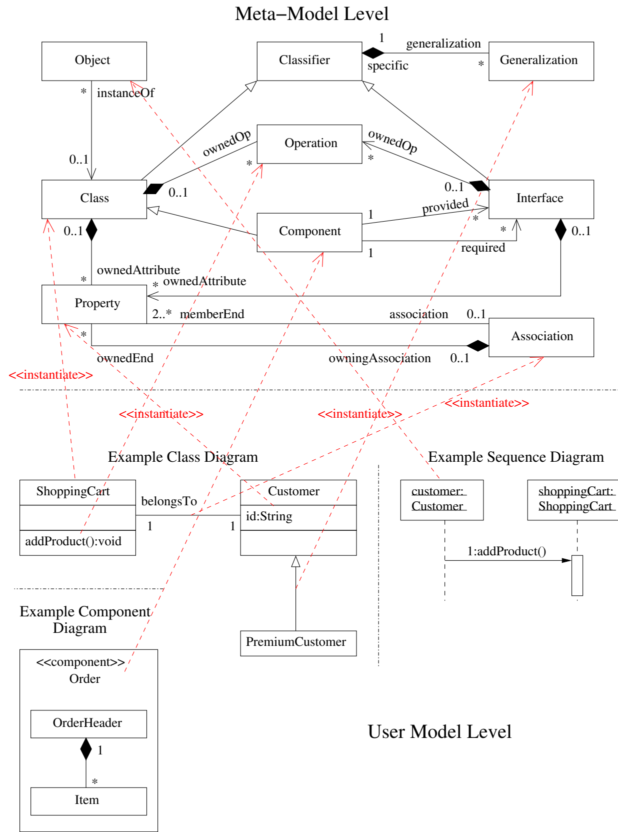

Fig. 1. Relationship between the UML metamodel and user models

Figure 2 shows the translation into DL of a small part of the UML metamodel seen in figure 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 `ModelElement`, 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 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`.

Figure 3 shows the translation into DL of a part of the class diagram shown in figure 1, 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". The second set of definitions corresponds to the translation of the class `Customer`. First, its name is set to the value "Customer". It is then related to the `Model` instance `inst-model1`, as the `Customer` class belongs to `model1`. Extra details about the `Customer` class are stored - in this case, the fact that the 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. *Tbox* for figure 1

is concrete and that it is a leaf (that is, it does not have descendants). The last set of definitions corresponds to the translation of the object `shoppingCart`. The name of the object instance is set to the value "shoppingCart". This object also belongs to `model1`, so it is also related to `inst-model1` using the `owned-element` role. Finally, the relationship between the class `ShoppingCart` and the instance `shoppingCart` is registered, relating the instances representing the two model elements using the `instance-of` role.

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

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

; Instance representing the shoppingCart object
(instance inst-shoppingCart object)
(constrained inst-shoppingCart name-of-shoppingCart name)
(constraints (string= name-of-shoppingCart "shoppingCart"))
(related inst-model1 inst-shoppingCart owned-element)
(related inst-shoppingCart inst-ShoppingCart instance-of)
```

Fig. 3. Part of the *Abox* (for figure 1)

The following theorems prove that DL is expressive enough to represent all well-formed user-defined UML models.

Theorem 1 *The UML metamodel can be completely described in terms of description logics.*

Proof. The UML metamodel is completely defined in terms of class diagrams [22]. And, according to [3], class diagrams can be completely described in terms of description logics.

□

Theorem 2 *All user defined UML models can be completely described in terms of description logics.*

Proof. All well formed user-defined UML models are instantiations of the UML metamodel [23]. And by Lemma 1, the UML metamodel can be completely described in terms of description logics.

□

So, the *Tbox* and the *Abox* can be built using the information contained in the UML metamodel and the user defined model, respectively. Checks are implemented using queries to retrieve sets of objects that obey certain conditions.

3 MCC

MCC (Model Consistency Checker) is a visual tool that allows consistency checking at the UML level. MCC uses DL as a means for consistency checking hiding the formalism, so all that the user sees are UML models. 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 only deal with UML diagrams will make it easier to maintain evolving systems, as designers only focus on the modeling level of abstraction.

3.1 Implementation Issues

Figure 4 shows the three components that make up the MCC tool:

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

This figure also shows the relationships that exist between MCC, Poseidon for UML and Racer.

The Visual Query Interface is implemented as a plug-in for Poseidon for UML 2.0. This CASE tool provides a graphical interface where UML models

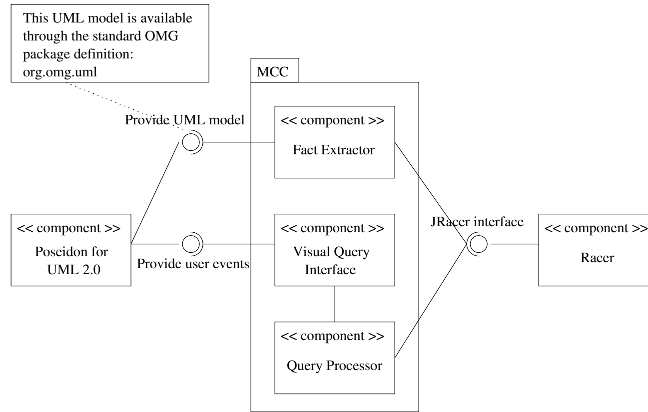

Fig. 4. MCC components

can be designed by dragging and dropping UML element templates like components and associations. This software has a large amount of users because, even though it is not opensource, it is free (the Community edition) and evaluation (the Standard edition) versions are available[2]. It is implemented in Java, which makes it completely portable. This software is robust, currently version 2.6, 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 an existing, known interface.

Racer [11] was chosen as the DL tool to be used. The reasons for this choice is that it possesses a very expressive concept definition language and it also offers efficient reasoning algorithms based on tableau calculus. This system implements the description logic $\mathcal{ALCQHIR}+$ also known as $\mathcal{SHIQ}$ (see [15]). This is the basic logic $\mathcal{ALC}$ augmented with qualifying number restrictions, role hierarchies, inverse roles and transitive roles. In addition to these basic features, Racer also provides facilities for algebraic reasoning including concrete domains for dealing with:

- min/max restrictions over the integers,
- linear polynomial (in)equations over the reals or cardinals with order relations,
- nonlinear multivariate polynomial (in-)equations over complex numbers,
- equalities and inequalities of strings.

Racer also supports the specification of general terminological axioms. A *Tbox* may contain general concept inclusions (GCIs), which state the subsumption

[2] The plug-in requires the Standard edition, version 2.5, since plug-ins can not be loaded into the Community version.

relation between two concept terms. Multiple definitions or even cyclic definitions of concepts can be handled by Racer.

The instance query mechanism has been improved, offering not only an API of functions for querying the *Tbox* and *Abox*, but also a flexible instance retrieval mechanism, allowing the construction of complex queries with placeholders [12], which will be used in MCC to implement the consistency checks. Another advantage is its Client / Server architecture, which allows easy access from other applications. Racer also offerers an extensible Java client interface, JRacer, making it straightforward to implement interactions between the reasoning engine and Java-based user interfaces. The Fact Extractor and Query Processor components are implementations of this interface.

After the user has loaded the MCC plug-in into the Poseidon framework[3], any model loaded into Poseidon will be loaded into Racer using the Fact Extractor component. Intra-model consistency analysis can then be applied by selecting consistency problems to detect from a pre-established list of implemented detection predicates. These are available in a user-friendly manner, and the user will never manipulate DL predicates directly. When applying a predicate, the Query Processor will communicate with Racer, which will reason using the facts already provided by the Fact Extractor, and this component translates the Racer output into user-readable information. Later versions will offer inconsistency solutions whenever possible, directly manipulating the user-created model.

3.2 Design Decisions

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 temporal 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 [10]. Our decision has two direct consequences: the user is always in control and the implementation of the tool is simpler.

Model loading from Poseidon to Racer and model checking are independent. This allows the user to reload the model when major changes are introduced. Any number of consistency checks can then be applied without the extra overhead of re-translating the model between checks. An XML file stores the plug-in's configuration, identifying which checks are implemented and by which classes. This allows easy extension of the set of checks that the tool supports, as reflexion is used to load the classes responsible for the individual checks. The detection predicate only inform the user about the inconsistencies that were found. It is up to the user to decide if an inconsistency is deliberate or if it has to be solved.

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,

[3] Using the Plug-in menu provided by Poseidon.

that returns the translation into DL of the object. The translator instance uses the model element's dynamic type in order to determine which individual translation method should be invoked. This allows for the seamless integration of translation methods for new element types and the application of changes to existing translations.

3.3 Existing Functionality

A set of 18 consistency relationships not forced by the metamodel definition has been studied in [26]. From this list, five consistency checks are already implemented as part of MCC: abstract object, incompatible behavior (state vs. sequence diagrams), multiplicity, classless instance and observable behavior conflicts (state vs. state). 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. 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. Multiplicity conflicts arise when a link between objects in a sequence diagram does not respect the multiplicity restrictions imposed by the corresponding association in the class diagram. Classless instance conflicts arise when an object in a sequence diagram is the instance of a class that does not exist in any of the class diagrams of the model. Finally, observable behavior refers to the fact that after all new events have been hidden, each trace of the subclass state diagram should be contained in the set of traces of the superclass state diagram.

The definition of these consistency relationships originally included in [26] had to be modified so as 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.

Other relevant changes introduced to the UML 2.0 specification are with respect to component-based development and activities, neither of which were covered by the previous set of predicates. In order to test the extensibility of the framework, we experimented with including support for consistency checking involving component diagrams. In order to include new diagram elements, the model translation facility had to be extended, including the translation rules for the new elements. This was achieved in a straightforward manner, treating the new elements like the rest of the translated UML metamodel. The new check had to be included in the XML that configures the available checks. Finally, the check was implemented by extending the established interface for consistency checks. This resulted in the seamless integration of a new type of consistency check, including reasoning about new diagrams.

4 Software Architecture Specification using UML

Object-orientation has been one of the most popular paradigms of the last two decades as it promotes good software engineering practices as separation of con-

cerns, encapsulation and inheritance, and reuse. First, object-oriented programming languages appear, but soon it was clear that a methodology or a guideline for supporting the complete life cycle was also required. UML had a big success as an object-oriented design notation mainly because there was an evident need for it in the object-orientation community, and also because it brought a standard notation everybody involved in the design process could agree on. UML served its purpose successfully, and a big user community adopted it as *the* standard design notation.

Meanwhile, the characteristics of the systems being developed continue evolving, growing in size and complexity, and fine grain objects started to be too small as the modeling abstraction for the new setting. Component-based development appeared as the new design paradigm and new concepts were involved. Also, software architecture matured in the last decade as an independent research area within software engineering. It is mainly concerned with the definition of components, interfaces, interaction between components, connectors, and configurations. New languages appeared for dealing with architectural specifications: architecture description languages (ADLs); they have been created and used almost exclusively in academia.

Software engineering practitioners had to face these new challenges with the available tools, so UML started to be commonly used for architectural specification even though it lacked the required constructs [9]. Thus, the new UML 2.0 standard [23] included some elements that basically add semantics to already existing component diagrams. In this way, components can not only define a set of interfaces of provided services, but also a set of interfaces of required services. Moreover, now component interfaces –named ports– are defined as stereotyped classes whose behavior can be specified using already existing state diagrams. These new features enhance the expressive power of UML as an ADL [20].

4.1 Consistency in Architectural Specifications

Component-based development allows designers to reason about systems in a high level of abstraction. Each component or subsystem has clearly defined responsibilities and they interact through a well established interface. Systems are specified as possibly small sets of interacting components. Classifiers that form part of a component should be highly coupled and must work together in order to provide the services that the component offers to other components. A component can also require services from other components in order to carry out its responsibilities. Classifiers belonging to different components should never be directly associated - they should always be indirectly associated through component required and provided interfaces. If component interfaces are strictly obeyed, old component implementations can be removed and replaced by new component implementations without repercussions in the rest of the system. So, respecting component required and provided interfaces is important when building evolvable component-based systems. This is especially true if the components are developed independently.

In order to offer a service through an interface, a classifier belonging to the component should implement the service as a public operation. The UML specification does not force this consistency. The reason for this is that component diagrams are usually built in a top-down fashion, that is, the services are first established, determining the internal classifier who is responsible for each at a later point in time. Designers usually divide the system into logical components, deciding what responsibilities each component will have and the dependencies between them. This diagram is later refined as classifiers are added to the components. As classifiers are added, the designers should be careful to remember to add the offered services as public operations of the necessary classifiers. If these diagrams become too complex, it is easy to forget to add an operation, or if a service changes, to propagate these changes into the classifiers that belong to the component. If no class belonging to the component implements a service offered by a component, or if it is a private or protected operation, it is said that the service offered by the component is not available.

Definition 1. *There is a service not available inconsistency in a UML model whenever there are services offered as part of one of its components offered interface that are not public operations implemented by any of the component's internal classifiers.*

MODEL(m) - m is a UML model
COMP(m) - set of all components that are part of model m
COS(i) - set of component i offered services
Class(i) - set of classifiers internal to component i
ClPO(j) - set of public operations of classifier j

$$\begin{aligned} Service\ not\ available(m) \Leftrightarrow\ & MODEL(m) \wedge \\ & \exists i \mid i \in COMP(m) \wedge \\ & COS(i) \not\subseteq \bigcup_{j \mid j \in Class(i)} ClPO(j) \end{aligned}$$

We are assuming that offered services of a component are elements of the same type as classifier operations.

4.2 Inconsistency detection example

We now show how MCC can be used to detect the Service Not Available inconsistency. The component diagram shown in figure 5 is taken from the UML 2.0 superstructure specification [22, 23]. The component `Order` has two internal classifiers, `OrderHeader` and `LineItem`. In this model, no operations have been specified yet for these classes, but `OrderEntry`, a offered interface for `Order`, offers the operation `getOrderEntry`. This is a Service Not Available inconsistency, as none of the internal classifiers implement this operation.

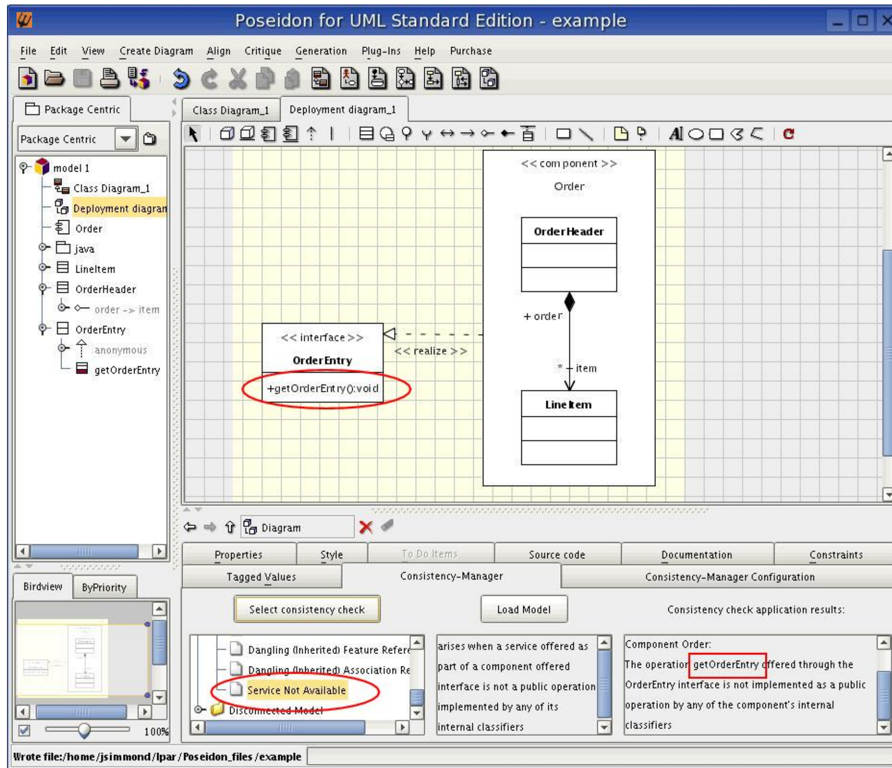

Fig. 5. Example of Service Not Available inconsistency

5 Related Work

A wide range of different approaches for checking consistency in UML models has been proposed in the literature. Engels et al. [7] motivate 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. This technique only deals with statechart inheritance problems, and users must be familiar with CSP. Statechart inheritance problem detection will be offered by MCC in a future version, as it has already been proved in [26] that DL can be used to detect this kind of problem. Ehrig and Tsiolakis [6] 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. Again, the user must be familiar with another formalism which is used to deal with only one family of consistency problems.

MCC offers a uniform manner in which to deal with a seamlessly scalable set of consistency problems.

The problem of verifying whether the interactions expressed by a collaboration diagram can indeed be realized by a set of state machines, has been treated by Schfer et al. [25]. They have developed HUGO, a prototype tool that checks if state machines (compiled into PROMELA [13] 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. [19], 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 and the user is not entirely familiar with the formalism. The BVUML tool skips the user translation problem. MCC also hides any translations from the user, and it also uses UML 2.0 metamodel definitions in order to determine consistency problems.

Rational Rose [17], another popular UML CASE tool, also incorporates adhoc model consistency checking. This checks the whole model, applying all the checks available. The user cannot specify which consistency checks are to be 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 dumped 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 broken links between model elements (for example, missing class, broken association, etc.) and model statistics and does not provide more complex checks like incompatible behavior or service not available.

One of the most common problems regarding framework evolution includes how to evolve its architecture without impacting the previously created applications. Due to the variety of methodologies used for modeling system architectures, there is very little tool support for framework evolution. For example, in [5], a profile for UML (UML-F) is introduced and used to assist framework maintenance and evolution, but no tool support is provided yet. On the other hand, using UML 2.0 a user can model system architectures using mainly class and component diagrams. MCC can then be used to check in an automated manner for framework and component inconsistencies, like the availability of services that are offered by a component.

Using architecture description languages (ADL) to formally describe system architectures is usually reserved for critical systems, as applying ADLs is a long and complex process, requiring profound knowledge of ADL specifications. Also, most ADL work is academic. There are various ADLs, so there is also the problem of communicating and comparing specifications written in different ADLs. UML 2.0 provides specialized constructs for architecture specification and, although it is still not necessarily the most expressive ADL, it is powerful if we consider that architectural specifications in UML are naturally integrated with detailed

design, regardless of the user-friendliness of graphical specifications ant the vast application experience not only in academia, but also in industry.

6 Conclusion

Due to the size and complexity of modern systems, component-based design has become popular. ADLs exist as a form of specifying architectures and components, but this approach is rarely used in real industrial applications, as these languages are highly formal, difficult to apply and hard to understand. Box-and-line diagrams are popular, but lack formality. UML is the de facto standard language for software modeling. Its newest version -UML 2.0- introduced new features for documenting software architectures as part of the standard. This standard specifies what can be modeled using UML and how, but it does not enforce consistency, as temporal inconsistencies are sometimes desired. It is possible to cross-check these diagrams by hand, but it is an error-prone process because of the model's size and complexity. Thus, there is a need for automated model consistency checks.

We offer MCC, a framework that provides UML 2.0 model checking using automated reasoning provided by a DL implementation. It already includes six different consistency checks. MCC is an extensible tool that can incorporate new checks, implemented as a plug-in integrated into an existing UML 2.0 CASE tool, 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 Racer 1.7 [11] behind the scenes. Both the UML CASE tool and the DL engine are robust systems. The whole configuration is portable, as it is all implemented in Java (including Poseidon and Racer). It is a user-friendly tool that has solid theoretical foundations.

This demonstrates that DL can provide a means for translating the UML metamodel, the user-defined models and the consistency checks as the *Tbox*, *Abox* and queries over the previous knowledge, respectively. DL tools are also sufficiently mature and efficient, providing APIs for easy integration with other tools. This allows the use of logics in areas outside traditional logic domains like machine learning and AI, adding formality to areas that were in need of it, like automated UML model consistency checking.

6.1 Ongoing Work

The currently available version of MCC is an academic prototype, providing a reduced list of implemented consistency checks. The remaining checks mentioned in [26] must also be added to the tool. New checks concerning other diagrams can also be included, as well as translating new diagrams. The UML 2.0 relationship between interfaces and protocol state machines is ideal as the next candidate for study and inclusion into the MCC framework. Currently, only academic examples

have been used to test the tool. Testing with larger examples must be carried out, for example, building a benchmark using known industrial examples.

Finally, taking advantage of the fact that the user models are already translated into a formalism and reasoning activities can be carried out, we can also implement refactorings [8]. Refactorings are behavior-preserving transformations that may be applied to design and implementation artifacts, in order to restructure and clean designs. Currently, there is tool support for code-level refactorings [16, 24] but this is language-dependent. The experiments carried out in [26] show that this idea is feasible applying the same strategy.

References

1. Carlos Areces. *Logic Engineering. The Case of Description and Hybrid Logics.* PhD thesis, ILLC, University of Amsterdam, 2000.
2. F. Baader, D. McGuinness, D. Nardi, and P.F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications.* Cambridge University Press, 2003.
3. Andrea Calì, Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. A formal framework for reasoning on uml class diagrams. In *Proc. of the 13th Int. Sym. on Methodologies for Intelligent Systems (ISMIS 2002)*, volume 2366 of *Lecture Notes in Computer Science*, pages 503–513. Springer, 2002.
4. Paul Clements, Felix Bachman, Ian Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stofferd. *Documenting Software Architectures. Views and Beyond.* Addison-Wesley, 2002.
5. Mariela Cortés, Marcus Fontoura, and Carlos de Lucena. Using Refactoring and Unification Rules to Assist Framework Evolution. *UPGRADE*, 5(2):49–55, April 2004.
6. H. Ehrig and A. Tsiolakis. Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In H. Ehrig and G. Taentzer, editors, *ETAPS 2000 workshop on graph transformation systems*, pages 77–86, March 2000.
7. Gregor Engels, Reiko Heckel, and Jochen Malte Küster. Rule-based specification of behavioral consistency based on the UML meta-model. In Martin Gogolla and Cris Kobryn, editors, *Proc. Int'l Conf. UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, number 2185 in Lecture Notes in Computer Science, pages 272–286. Springer-Verlag, October 2001. Toronto, Canada.
8. Martin Fowler. *Refactoring: Improving the Design of Existing Programs.* Addison-Wesley, 1999.
9. David Garlan, Shan-Wen Chen, and Andrew J. Kompanek. Reconciling the Needs of Architectural Description with Object-Modeling Notations. *Science of Computer Programming Journal, Special issue on UML*, 44(1):23–49, July 2002.
10. John C. Grundy, John G. Hosking, and Warwick B. Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Transactions on Software Engineering*, 24(11):960–981, 1998.
11. Volker Haarslev and Ralf Möller. RACER, April 8 2003. `http://www.fh-wedel.de/~mo/racer/`.
12. Volker Haarslev, Ralf Möller, Ragnhild Van Der Straeten, and Michael Wessel. Extended Query Facilities for Racer and an Application to Software-Engineering

Problems. In *Proceedings of the International Workshop in Description Logics 2004 (DL2004)*, 2004.
13. G.J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.
14. Ian Horrocks. FaCT, September 2004. `http://www.cs.man.ac.uk/~horrocks/FaCT/`.
15. Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Reasoning with individuals for the description logic $\mathcal{SHIQ}$. In David McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction (CADE-17)*. Springer-Verlag, 2000.
16. IBM. Eclipse, October 2004. `http://www.eclipse.org`.
17. IBM. Rational Software, October 2004. `http://www-306.ibm.com/software/rational/`.
18. ISI. Loom, April 2003. `http://www.isi.edu/isd/LOOM/LOOM-HOME.html`.
19. Boris Litvak, Shmuel Tyszberowicz, and Amiram Yehudai. Consistency Validation of UML Diagrams. In *Correctness of Model-based Software Composition (CMC) Workshop, ECOOP*, 2003.
20. Nenad Medvidovic and Richard Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):73–90, January 2000.
21. Michael Moors. Rose Model Checker, October 2004. `http://www.rationalrose.com/modelchecker/index.htm`.
22. Object Management Group. UML 2.0 Infrastructure Specification. 03-09-15.pdf, September 2003. `http://www.omg.org/docs/ptc/`.
23. Object Management Group. UML 2.0 Superstructure Specification. 04-05-02.pdf, May 2004. `http://www.omg.org/docs/ptc/`.
24. Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.
25. T. Schfer, A. Knapp, , and S. Merz. Model Checking UML State Machines and Collaborations. *In Electronic Notes in Theoretical Computer Science*, 47:1–13, 2001.
26. Jocelyn Simmonds. Consistency Maintenance of UML Models with Description Logics. Master's thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium and Ecole des Mines de Nantes, France, 2003.