UML2Alloy: A Challenging Model Transformation

Kyriakos Anastasakis[1], Behzad Bordbar[1], Geri Georg[2], and Indrakshi Ray[2]

[1] School of Computer Science, University of Birmingham, Edgbaston, Birmingham, UK
{K.Anastasakis,B.Bordbar}@cs.bham.ac.uk

[2] Computer Science Department, Colorado State University, Fort Collins, Colorado, USA
{georg,iray}@cs.colostate.edu

Abstract. Alloy is a formal language, which has been applied to modelling of systems in a wide range of application domains. It is supported by Alloy Analyzer, a tool, which allows fully automated analysis. As a result, creating Alloy code from a UML model provides the opportunity to exploit analysis capabilities of the Alloy Analyzer to discover possible design flaws at early stages of the software development. Our research makes use of model based techniques for the automated transformation of UML class diagrams with OCL constraints to Alloy code. The paper demonstrates challenging aspects of the model transformation, which originate in fundamental differences between UML and Alloy. We shall discuss some of the differences and illustrate their implications on the model transformation process. The presented approach is explained via an example of a secure e-business system.

1 Introduction

The Unified Modelling Language (UML) [1] is the de-facto language used in the industry for specifying the requirements and the design of software systems. Detecting faults during the early stages of the software development lifecycle, instead of the later stages, provides a significant saving in cost and effort. This necessitates analysing the requirements and design specifications for potential errors and inconsistencies before the system has been developed. Manual analysis is error-prone and tedious. A number of approaches have been proposed in the literature [2,3,4] for analysing UML specifications. These analyses rely mainly on using theorem provers. Theorem provers are hard to use, require expertise, and the analysis requires manual intervention. Consequently, such approaches are not very suitable for use in real-world applications.

In this paper, we advocate the use of Alloy [5] for analysing UML specifications. Alloy is a modelling language for expressing complex structural constraints and behaviour. It has a well-designed syntax most suitable for Object Oriented modelling. Moreover, Alloy is supported by a software infrastructure [6], which provides fully automatic analysis of models in the form of simulation and checking the consistency of specifications. Alloy has received considerable attention in the research community. For example, it has been successfully applied to modelling and analysis of protocols in distributed systems [7], networks [8] and mission critical systems [9].

There are clear similarities between Alloy and UML languages such as class diagrams and OCL. From a semantic point of view both Alloy and UML can be interpreted by sets of tuples [5,10]. Alloy is based on first-order logic and is well suited for expressing constraints on Object Oriented models. Similarly, OCL has extensive constructs for expressing constraints as first-order logic formulas. Considering such similarities, model transformation from UML class diagrams and OCL to Alloy seems straightforward. However, UML and Alloy have fundamental differences, which are deeply rooted in their underlying design decisions.

For example, Alloy makes no distinction between sets, scalars and relations, while the UML makes a clear distinction between the three. This has grave consequences in the transformation between the two languages. The current state of model transformation techniques is not dealing with such issues. In this paper we reflect on such differences and their effect on the transformation.

We have incorporated the ideas presented in this paper in a tool called UML2Alloy. UML2Alloy which has been applied to the analysis of discrete event systems [11] and the architecture of enterprise web applications [12].

The next section provides an overview of basic concepts used in this paper.

2 Preliminaries

This section provides a brief introduction to the basic concepts of the MDA and Alloy, which will be used in the rest of the paper.

Model Driven Architecture: The method adopted in this paper makes use of Model Driven Architecture (MDA) [13] techniques for defining and implementing the transformations from models captured in the UML class diagram and OCL into Alloy. Central to the MDA is the notion of *metamodels* [14]. A metamodel defines the elements of a language, which can be used to represent a model of the language. In the MDA a model transformation is defined by mapping the constructs of the metamodel of a *source* language into constructs of the metamodel of a *destination* language. Then every model, which is an instance of the source metamodel, can be automatically transformed to an instance of the destination metamodel with the help of a model transformation framework [15].

Alloy: Alloy [5] is a textual modelling language based on first-order relational logic. An Alloy model consists of a number of *signature* declarations, *fields*, *facts* and *predicates*. Each signature denotes to a set of *atoms*, which are the basic entities in Alloy. Atoms are *indivisible* (they can not be divided into smaller parts), *immutable* (their properties remain the same over time) and *uninterpreted* (they do not have any inherent properties) [5]. Each field belongs to a signature and represents a relation between two or more signatures. Such a relation denotes to a set of tuples of atoms. In Alloy *facts* are statements, which define constraints on the elements of the model. Parameterised constraints, which are referred to as *predicates*, can be invoked from within facts or other predicates.

Alloy is supported by a fully automated constraint solver, called Alloy Analyzer [6], which allows analysis of system properties by searching for instances of the model. It is possible to check that certain properties of the system (*assertions*) are satisfied. This is achieved by automated translation of the model into a Boolean expression, which is analysed by SAT solvers embedded within the Alloy Analyzer. A user-specified *scope* on the model elements bounds the domain. If an instance that violates the assertion is found within the scope, the assertion is not valid. However, if no instance is found, the assertion might be invalid in a larger scope. For more details on the notion of scope, please refer to [5, Sect. 5].

One important characteristic of Alloy is that it treats scalars and sets as relations. For example, a relation between two atoms *A1* and *A2* is represented by the pair: $\{(A1, A2)\}$. A set like: $\{A1, A2\}$ is represented by a set of unary relations: $\{(A1), (A2)\}$. Finally a scalar, is represented as a singleton unary relation. For example, the scalar *A1*, will be represented in Alloy as: $\{(A1)\}$.

Treating both scalars and sets as relations, is an interesting property of Alloy, which makes it distinguishable from other popular modelling notations and particularly UML. Hence it introduces additional complexity into the definition of the transformation rules. The following section discusses our MDA based approach to transform UML class diagrams annotated with OCL constraints to Alloy.

3 Model Transformation from the UML to Alloy

This section presents a brief description of our work. We use an MDA compliant methodology to transform a subset of UML class diagram models enriched with OCL constraints to Alloy.

Figure 1 depicts an outline of our approach. Using the EBNF representation of the Alloy grammar [5], we shall first generate a MOF compliant [14] metamodel for Alloy. We then select a subset of the class diagrams [16] and OCL [17] metamodels. To conduct the model transformation, a set of transformation rules has been defined. The rules map elements of the metamodels of class diagrams and OCL into the elements of the metamodel of Alloy. The rules have been implemented into a prototype tool called UML2Alloy. If a UML class diagram, which conforms to the subset of UML we support is provided as input to UML2Alloy, an Alloy model is automatically generated by the tool.

The next section illustrates our work on transforming the EBNF representation of Alloy's grammar into a MOF compliant metamodel.

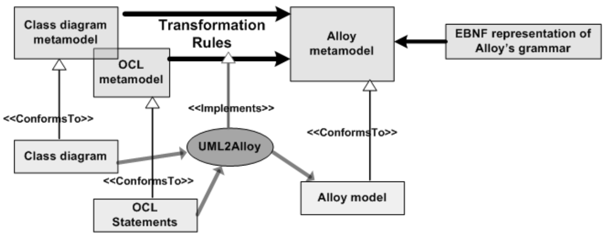

Fig. 1. Outline of the transformation method

3.1 EBNF to MOF

Alloy is a textual language and its syntax is defined in terms of its EBNF [18] grammar [5, Ap. B]. The grammar represents the concrete syntax of the Alloy language. In order to use the MDA, we need to convert the concrete syntax of the Alloy language to a MOF compliant abstract syntax representation. Wimmer and Kramler [19] have already proposed a method for generating a metamodel of a language, based on the EBNF representation of its syntax. We utilised their approach with some simplifications, since some of their proposals were not required in the case of Alloy. For example, we did not use annotations to give additional semantics to the Alloy metamodel that was generated.

Figure 2 depicts a portion of the Alloy metamodel we constructed for signature declarations. A signature declaration (*SigDecl*) is an abstract metaclass. It can either be an *ExtendSigDecl* or an *InSigDecl*, used for subtyping and subsetting signatures respectively. A *SigDecl* has a signature body (*SigBody*). It can contain a sequence of constraints (*ConstraintSequence*). A signature declaration can also specify a number of declarations (*Decl*). Declarations are used to define signature fields. They declare one or more variables (*VarId*) and are related to a declaration expression (*DeclExp*). A declaration expression can either declare a binary relation between signatures (*DeclSetExp*) or a relation that associates more than two signatures (*DeclRelExp*). Similarly, we have defined the parts of the Alloy metamodel which represent *expressions*, *constraints* and *operations*.

Since the construction of the metamodel was an intermediate step to utilise the MDA technology, we did not use OCL to specify well-formedness rules on the elements of the metamodel, an approach which is adopted by the UML specification. Instead well-formedness rules were embedded in the transformation, ensuring that the generated Alloy models are well-formed.

3.2 Mapping Class Diagram and OCL to Alloy

This section presents a brief introduction on the transformation rules from UML to Alloy. It provides an informal correspondence between elements of the UML and Alloy metamodels, as a basis on which to present the challenges of the transformation. A more detailed description of the transformation rules can be found in [11]. Due to space limitations the UML and OCL metamodels are

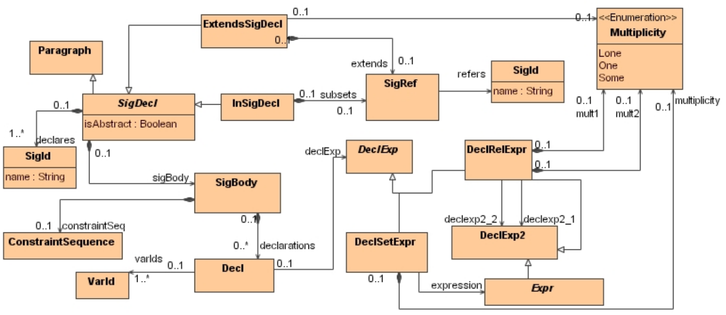

Fig. 2. Part of the Alloy metamodel used to represent signature declarations

not presented here. An extensive explanation can be found in the respective specification documents [16, p. 29] [17].

Table 1 provides an informal correspondence between the most crucial elements of the UML and OCL metamodels and Alloy. More specifically a UML *Class* is translated to an Alloy signature declaration (*ExtendsSigDecl*), which defines a *SigId* with the same name as the class name. If the class is not a *specialization* the Alloy signature is not related to any *SigRef*. Otherwise it will be related to a *SigRef*, which references the signature it might extend.

For example, the *Client* class in the UML model of Fig. 3 is transformed to an *ExtendsSigDecl*, which *declares* a *SigId*, whose *name* is *Client*. Because it doesn't represent a subclass, it is not related to any *SigRef*. Similarly the *SoftwareClient* and *WebClient* are transformed to an *ExtendsSigDecl*. Unlike the *Client* class though, they are related to a *SigRef*, which refers to the *SigId* generated to represent the *Client* class.

The next section presents an example UML class diagram, which will be used to illustrate the challenges of the transformation from UML to Alloy.

Table 1. Informal mapping between UML and Alloy metamodel elements

UML+OCL metamodel element	**Alloy metamodel element**
Class	ExtendsSigDecl
Property	DeclExp
Operation	Predicate
Parameter	Decl
Enumeration	ExntedsSigDecl
EnumerationLiteral	ExtendsSigDecl
Constraint	Expression

4 Example UML Class Diagram

Figure 3 depicts a UML class diagram that represents the login service of an e-commerce application. The e-commerce system allows clients (i.e. Client) to

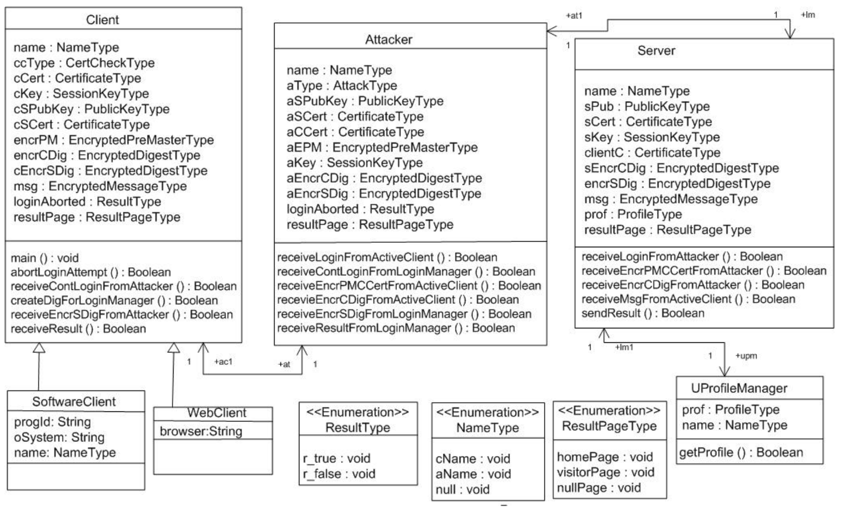
Client
name : NameType
ccType : CertCheckType
cCert : CertificateType
cKey : SessionKeyType
cSPubKey : PublicKeyType
cSCert : CertificateType
encrPM : EncryptedPreMasterType
encrCDig : EncryptedDigestType
cEncrSDig : EncryptedDigestType
msg : EncryptedMessageType
loginAborted : ResultType
resultPage : ResultPageType
main () : void
abortLoginAttempt () : Boolean
receiveContLoginFromAttacker () : Boolean
createDigForLoginManager () : Boolean
receiveEncrSDigFromAttacker () : Boolean
receiveResult () : Boolean
Attacker
name : NameType
aType : AttackType
aSPubKey : PublicKeyType
aSCert : CertificateType
aCCert : CertificateType
aEPM : EncryptedPreMasterType
aKey : SessionKeyType
aEncrCDig : EncryptedDigestType
aEncrSDig : EncryptedDigestType
loginAborted : ResultType
resultPage : ResultPageType
receiveLoginFromActiveClient () : Boolean
receiveContLoginFromLoginManager () : Boolean
receiveEncrPMCCertFromActiveClient () : Boolean
recevieEncrCDigFromActiveClient () : Boolean
receiveEncrSDigFromLoginManager () : Boolean
receiveResultFromLoginManager () : Boolean
Server
name : NameType
sPub : PublicKeyType
sCert : CertificateType
sKey : SessionKeyType
clientC : CertificateType
sEncrCDig : EncryptedDigestType
encrSDig : EncryptedDigestType
msg : EncryptedMessageType
prof : ProfileType
resultPage : ResultPageType
receiveLoginFromAttacker () : Boolean
receiveEncrPMCCertFromAttacker () : Boolean
receiveEncrCDigFromAttacker () : Boolean
receiveMsgFromActiveClient () : Boolean
sendResult () : Boolean
+at1
1
1
+lm
1
+lm1
1
+upm
1
+ac1
+at
1
SoftwareClient
progId: String
oSystem: String
name: NameType
WebClient
browser:String
<<Enumeration>>
ResultType
r_true : void
r_false : void
<<Enumeration>>
NameType
cName : void
aName : void
null : void
<<Enumeration>>
ResultPageType
homePage : void
visitorPage : void
nullPage : void
UProfileManager
prof : ProfileType
name : NameType
getProfile () : Boolean

Fig. 3. Partial model of the SSL protocol included in the e-commerce system login service

```
context Client::abortLoginAttempt (): Boolean
post abortLoginAttempt:
    self.loginAborted = ResultType::r_true and
    self.resultPage = ResultPageType::nullPage
```

Fig. 4. OCL specification of the abortLoginAttempt operation of the *Client* class

purchase goods over the internet. It is therefore susceptible to various attacks, including a man-in-the-middle attack that allows an attacker to intercept information that may be confidential. The login service has therefore been augmented with the SSL (Secure Sockets Layer [20]) authentication and confidentiality protocol. We model the man-in-the-middle attack in our example by adding an Attacker class that intercepts all communications between the Client and the e-commerce server, Server, possibly changing message content prior to passing them onto the intended recipient. The SSL protocol works the same whether the Client is a SoftwareClient or a WebClient, but the SoftwareClient provides some extra functionality to the user. If the SSL handshake completes successfully, a secret session key that can be used for message encryption and decryption will have been exchanged between Client and Server. All further communication between them will be encrypted, and thus confidential. If the handshake fails, all communication is aborted between Client and Server.

Figure 3 depicts a high level representation of the system, where attributes of the classes hold the values of the messages exchanged between the entities that participate in the interactions. Due to space limitations some user defined types, such as the *EncryptedDigestType* are not represented in this diagram. For an extended study of this model, please refer to [21].

Figure 4 depicts an excerpt of the OCL specification of the *abortLoginAttempt* method, which will be used later on to demonstrate the differences between OCL and Alloy.

5 Differences Between UML and Alloy Which Influence the Transformation

Although both UML and Alloy are designed to be used in Object-Oriented (OO) paradigm, the two languages have different approaches to some of the fundamental issues of OO such as inheritance, overloading and predefined types [22]. Some of these differences directly influence the model transformation process. In this section, we shall discuss such differences and explain how our approach deals with them.

Inheritance: Both UML and Alloy support inheritance. In the UML, a child class inherits and can specialise the properties of one or more classes [16, p. 126]. The UML standard uses the term '*redefines*' to denote attribute or operation overriding.

In Alloy a *signature* can extend another signature and the elements of the subsignature are a subset of the elements of the supersignature. However, a subsignature can't declare a field whose name is the same as the name of a field of its supersignature. Thus a subsignature can't override the fields of a supersignature. In our transformation we have resolved this shortcoming, by renaming all Alloy fields which have naming conflicts. This is explained with the help of an example.

Consider the SoftwareClient class of Fig. 3. It has an attribute *name*, which overloads the attribute *name* of the Client class. In order to transform this model, we need to create a field with a unique name in Alloy and change all references, which refer to the *name* attribute of the SoftwareClient class, to reference the uniquely named field. However, this brings additional complications. According to the UML specification constraints of a superclasses are propagated to the subclasses. In particular it is mentioned that: '*A redefining element... can add specific constraints or other details that are particular to instances of the specializing redefinition context that do not contradict invariant constraints in the general context.*' [16, p. 126]. Let us assume the following constraint: *self.name* <> *NameType::null* exists in the Client class and the constraint *self.name* <> *NameType::aName* exists in the SoftwareClient. During the transformation the *name* attribute of the SoftwareClient is renamed to *name1*. The original constraint is then translated to a signature constraint: *name1* ! = *aName* in Alloy. However, this allows for the field *name1* to have a *null* value, which is not acceptable in the original UML model. Therefore another constraint (i.e. *name1* ! = *null*) needs to be injected in the translated Alloy model to reflect the constraints applied to the *name* attribute of the *Client* class. A similar approach is followed when dealing with operation overriding.

Namespace: All UML model elements are defined in a *namespace* [17, p. 72]. For example, classes in a class diagram are usually defined in the namespace of the package, while attributes are defined in the namespace of the class they belong to.

Model elements of an Alloy model also belong to a namespace [5, p. 254]. However, the notion of a namespace in Alloy and UML are slightly different. For example, the UML specification defines that: '*The set of attribute names and class names need not be disjoint*' [17, p. 178]. In Alloy on the other hand signature names, have to be distinct from their field names.

Therefore we need to ensure that during the transformation a unique name is created for Alloy elements that belong to the same namespace. In our approach we first identify elements in the UML metamodel, which belong to different UML namespaces, but are translated to the same Alloy namespace (e.g. class operations). If those elements do not have a unique name, we make sure to assign to them a unique name during the transformation to Alloy. All references to those elements in the original UML model, are changed during the transformation to reference the unique names in the generated Alloy model.

Another issue is that in OCL the instance of the class on which the operation is applied, can be accessed using the *self* keyword. In Alloy there is no such concept for predicates, making it difficult to reference the instance of the signature on which the Alloy predicate is applied. As a solution we pass the instance as a parameter to the predicate. For an example of a solution to this problem, consider the *abortLoginAttempt()* operation of the Client. Its OCL specification makes use of the *self* keyword and is depicted in Fig. 4. Following our transformation rules, we translate it in Alloy to the following predicate, where *act*, which is an instance of the signature Client, is passed as a parameter to the predicate:

```
pred abortLoginAttempt(act:Client){
act.loginAborted = r_True && act.resultPage = nullPage }
```

Sets, Scalars, Relations and Undefinedness: Alloy treats sets and scalars as relations. In particular in Alloy a relation denotes to a set of tuples. The number of elements in each tuple depends on the arity of the relation. For example, a binary relation is represented by a 2-tuple. A set is represented as a unary relation and a scalar is a singleton unary relation [5, p. 45].

In UML on the other hand, sets and scalars have the standard meaning they have in set theory. The equivalent of relations in UML is an association between classes, which is represented as a set of tuples [17, p. 184].

These differences in the two languages stem from the fact that UML and Alloy have different design philosophies. More specifically one of the purposes of UML is to represent Object Oriented programming concepts, where the distinctions between scalars and sets is clear. On the other hand Alloy was designed for analysing abstract specifications and the uniform way it deals with sets, scalars and relations contributes to its succinct syntax and leverages its expressiveness [23].

To explain this consider the navigation dot (.). In Alloy it is treated as the relational join [5, p. 59]. As a result navigating over an empty relation denotes to an empty set. Consequently Alloy doesn't need to address the problem of partial functions by introducing a special *undefined* value, like in UML [17, Ap. A.2.1.1]. More specifically let us assume in the model of Fig. 3 we have the following OCL statement.

$$\textit{context Client inv: self.at.name} = \textit{self.at.lm.name} \qquad (1)$$

In UML if the instance of the Client in which this OCL invariant is evaluated is related to no Attacker, the part *self.at.name* of this statement will denote to an *undefined* value. The result of the invariant will then be *undefined.* In an equivalent Alloy model, however, if the Client was related to no Attacker, such a constraint would always denote to true! This is because the left hand side part of the expression, that is *self.at.name*, would denote to an empty set relation. Similarly the right hand side, *self.at.lm.name* would evaluate to an empty set. Therefore the invariant will always evaluate to true.

This has serious implications because the statement will produce a different outcome in Alloy than in OCL. To overcome the problem, we check if an OCL statement can evaluate to an undefined value. To check for undefinedness, the *oclIsUndefined()* OCL operation is used. For example, the OCL statement of (1) will become:

```
context Client
    inv: if not self.at.name.oclIsUndefined() and
        not self.at.lm.name.oclIsUndefined() then
            self.at.name = self.at.lm.name
        else false  endif
```

In this case the modeller has specified that if any part of the expression is *undefined*, the invariant should evaluate to false. This ensures that the OCL statement will either evaluate to true or false, but not undefined. Such an expression is transformed to Alloy using our standard UML to Alloy transformation rules.

Predefined Types: The UML specification defines a number of primitive types (e.g. String, Real, etc.). Those types can be used when developing UML models. For example, the attribute *browser* of the *WebClient* class in Fig. 3 is of type String.

On the other hand, Alloy has a simple type system and the only predefined type it supports is Integers. However, some of the rest of UML's predefined types, can be modelled in Alloy. For example, a String, can be modelled as a sequence of characters and each character can be represented by an *atom*.

One consequence of this approach, is that while in UML primitive types and their operations are part of the metamodel, in Alloy they need to be defined on the model level (i.e. a String has to declared as an Alloy signature). Our transformation rules do this automatically for certain attribute types (e.g. String).

UML's extension mechanism: UML provides two extensions mechanisms [1, p. 11]. One is to create a *profile* and another one is to extend the UML metamodel. If UML has been extended, we need to incorporate the rules involving the new elements into the transformation.

Our current transformation deals with a subset of the standard UML and OCL metamodels. If the metamodels have been extended, the new semantics need to be incorporated into the transformation. For example, let's assume that UML has been extended with the ability to define a *Singleton* stereotype. This stereotype, when used on a class, restricts the class to have only one instance. This is expressed with the following invariant in OCL: *self.allInstances()* $\rightarrow$ *size() =1.* In such a case the transformation rules need to be adjusted accordingly. In particular, whenever a *Singleton* stereotype is found on a class, a constraint needs to be injected in the produced Alloy model, to impose that the transformed signature will have only one instance. The implementation of our transformation rules, is modular and uses the SiTra [15] transformation engine, which can be easily augmented to accommodate for any extensions.

Aggregation and Composition: The UML treats aggregation and composition as special kinds of associations [1, p. 112]. Alloy doesn't directly support notions like aggregation and composition. Fortunately [24] present a methodical way of refactoring aggregation and composition as an association with additional OCL constraints that represent the semantics of aggregation and composition. We utilise this approach, as it allows us to use transformation rules for binary associations and OCL constraints we have already defined.

Static vs Dynamic Models: Models in Alloy are static, i.e. they capture the entities of a system, their relationships and constrains about the system. An Alloy model defines an instance of a system where the constraints are satisfied. However, Alloy models do not have an inherent notion of states. In particular, Alloy does not have any built in notion of statemachine [5, Ap. B.5.1]

In UML the term 'static' is used to describe a view of the system, that represents the structural relations between the elements as well as the constraints and the specification of operations with the help of pre and post conditions. In UML, unlike Alloy, static models have an inherent notion of states. A *system state* is made of the values of objects, links and attributes in a particular point in time [17, p. 185].

Hence UML has an implicit notion of states, while Alloy does not support it directly. This introduces additional complexity in the transformation. Let us assume the following OCL statement is the definition of the *receiveResult()* operation of the Client:

```
context Client::receiveResult():void
pre: self.resultPage = ResultPageType::nullPage
post: self.resultPage = ResultPageType::homePage
```

To evaluate this expression two consecutive states are required, one to represent the state before the execution of the operation (precondition) and another to represent the state after the execution of the operation (postcondition). The OCL standard formally specifies the *environment* on which pre and postconditions are evaluated [17, p. 210].

If the specification of the *receiveResult()* operation, was directly translated to Alloy it would translate to:

```
pred receiveResult(act:Client){ act.resultPage = nullPage
act.resultPage = homePage }
```

However, such an Alloy specification leads to an inconsistent model. This is because the value *nullPage* and *homePage* are assigned to the *resultPage* field, at the same time. This leads to a logical inconsistency, as both statements can not be true (i.e. resultPage will either be the *nullPage* or *homepage*, but not both at the same time).

One solution, which has been proposed is to introduce the notion of a *state* at the model level [5,25]. This is a standard way of modelling dynamic systems in Alloy. Our approach uses this pattern of modelling dynamics in Alloy, to translate UML models to Alloy. This allows us to have two consecutive states and evaluate the preconditions of each operation on the first state, while evaluating the postcondition of an operation on the next state.

6 Analysis, Discussion and Future Work

This section presents a brief overview on the results of the analysis we conducted on the example UML model. It also provides a discussion on further issues that were encountered and suggests directions for future work.

6.1 Analysis Via Alloy

We applied our model transformation rules from UML to Alloy on the example model presented in Sect. 4. We checked the produced Alloy model, using the Alloy Analyzer. The assertion that must be validated is that if the Attacker obtains the secret session key, the handshake should always fail. This assertion can be specified using OCL:

```
context Client
inv sameKeySuccess: Client.allInstances() -> forAll(ac:Client |
    ac.loginAborted = ResultType::r_false implies (
    ac.cKey = SessionKeyType::symmKey and
    ac.at.sKey = SessionKeyType::symmKey
    and ac.at.aKey <> SessionKeyType::symmKey))
```

This OCL statement was automatically transformed to the following Alloy assertion:

```
assert sameKeySuccess{ all ac:Client | ac.loginAborted = r_false
implies (ac.cKey = symmKey && ac.at.lm.sKey = symmKey &&
ac.at.aKey != symmKey) }
```

This assertion was checked for a scope [5, p. 140] of six. A scope of six means that the Alloy Analyzer will attempt to find an instance that violates the assertion, using up to six instances for each of the entities defined in the class diagram of Fig. 3 (for example, Client, Attacker, Server). The assertion produced no counterexample, meaning that it is valid for the given scope.

6.2 Discussion and Future Work

This section briefly presents a discussion on further practical issues we had to deal with, when defining the transformation rules. Moreover it suggests directions for future work.

A difficulty that was encountered when defining the transformation rules, was that parts of the UML specification are inconsistent with the UML *informative semantics* section of the specification [17, An. A]. For example, even though the UML standard allows for overloading of attributes and operations [16, Sect. 7.3.46], the UML formal semantics part of the specification seems to adopt a different stance [17, p. 182]. In particular it doesn't allow for attributes or operations of a subclass to have the same name as the attributes and operations of its superclass. As explained in Sect. 5 we made use of the informal semantics in our transformation rules, since attributes and operations overriding is an important facility provided in object oriented modelling.

Another issue we had to overcome, originates in the nature of OCL. In particular the transformation rules had to be invoked recursively. For example, the definition of the abstract syntax of *If* expressions [17, p. 45] allows for any type of an OCL expression to be part of the *condition* clause. As a result, when defining the transformations someone needs to check the type of the *condition* expression and invoke the corresponding transformation rule, which will be used to transform that specific kind of expression to Alloy. This problem is dealt by the SiTra [15] transformation framework we used for our implementation. SiTra allows for recursive calls of transformation rules with dynamic type checking. Therefore depending on the type of the expression, the corresponding rule is automatically invoked.

The UML specification defines a number of concepts (e.g. *ordered* and *subsets* annotations of association ends, *package import*), which are not formally defined. For some of those concepts the semantics are not clear (e.g. package merge) [26]. Transformation rules for such concepts have not been defined yet. Modellers can define their interpretation of the semantics of such concepts on the model level using OCL.

While Alloy is a purely declarative language, OCL has an imperative flavour. For instance, the OCL standard allows for recursion: '*We therefore allow recursive invocations as long as the recursion is finite*' [17, p. 205]. Alloy on the other hand does not directly support recursion. It might be possible to represent some cases of recursion, using the expressiveness of the Alloy language as suggested in [22]. How this might be incorporated into our transformation rules in order to provide support for recursion, remains an issue for future research.

As we use Alloy to formalise UML, our approach admits some of the inherent limitations of the Alloy language. Some UML primitive types (such as Real numbers) can not be directly transformed to Alloy. Therefore it is not feasible to check whether certain properties involving real values are satisfied. Additionally the UML offers a number of collection types (e.g. *Sequence*, *Bag*), which can not be directly represented in Alloy. Moreover, since Alloy is a first-order language, it does not support nested collections. The possibility of representing the capabilities of UML collection types in Alloy remains to be investigated further.

7 Related Work

Formalising UML for the purpose of analysis is a popular approach. Evans et al [4] propose the use of Z [27] as the underlying semantics for UML. Marcano and Levy [2] advocate the use of B [28], while Kim [3] makes use of an MDA method to translate a subset of UML to Object-Z. These methods rely on theorem provers to carry out the analysis, which complicates the process.

A number of UML tools also provides support for analysis. For example, the USE tool (UML Specification Environment) [10] is a powerful instance evaluator with the ability of simulation.

Using Alloy to formalise UML has also received considerable attention. More specifically Denis et al [9] use Alloy to expose hidden flaws in the UML design of a radiation therapy machine. Georg et al [8] have used Alloy to analyse the runtime configuration of a distributed system. Unlike our work, those approaches conduct the translation from UML to Alloy manually, a procedure which is tedious and error prone.

Additionally there have been studies on the comparison of languages of UML and Alloy [22,29]. However, they do not use model driven approaches to demonstrate the differences.

Finally transformations from a three-valued logic language to a two-valued logic language, such as ours from UML to Alloy have been applied to the field of database semantics. For example, [30] propose the use of an interpretation operator to treat statements with undefined values in databases either as true or false.

8 Conclusions

Model transformations in the context of MDA are predominantly used for code generation. Model transformations can be used for the creation of analysable models, allowing the discovery of possible flows in the design of a system. Languages used for creation of analysable models have strong formal foundations. Hence, a model transformation from the UML to such languages is highly non trivial.

In this paper we have reflected on the lessons learned from the model transformation from UML class diagrams to Alloy. We discussed some of the differences between UML and Alloy. For example, the different perspectives on inheritance, functions, static and dynamic models. We also studied the implications of such differences in the model transformation. Our proposed solutions to such challenges were presented. The method is implemented in a prototype tool called UML2Alloy. The approach is illustrated with the help of an example from the security domain.

References

1. OMG: UML Infrastructure Document: formal/05-07-05, `http://www.omg.org`
2. Marcano, R., Levy, N.: Using B formal specifications for analysis and verification of UML/OCL models. In: Workshop on consistency problems in UML-based software development. 5th International Conference on the Unified Modeling Language, Dresden, Germany (2002)
3. Kim, S.K.: A Metamodel-based Approach to Integrate Object-Oriented Graphical and Formal Specification Techniques. PhD thesis, University of Queensland, Brisbane, Australia (2002)
4. Evans, A., France, R., Grant, E.: Towards Formal Reasoning with UML Models. In: Proceedings of the OOPSLA'99 Workshop on Behavioral Semantics (1999)
5. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, London, England (2006)
6. Jackson, D.: Alloy Analyzer website, `http://alloy.mit.edu/`
7. Taghdiri, M., Jackson, D.: A lightweight formal analysis of a multicast key management scheme. In: König, H., Heiner, M., Wolisz, A. (eds.) FORTE 2003. LNCS, vol. 2767, pp. 240–256. Springer, Heidelberg (2003)
8. Georg, G., Bieman, J., France, R.B.: Using Alloy and UML/OCL to Specify Run-Time Configuration Management: A Case Study. In: Evans, A., France, R., Moreira, A., Rumpe, B. (eds.) Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists. LNI, German Informatics Society, vol. P-7, pp. 128–141 (2001)
9. Dennis, G., Seater, R., Rayside, D., Jackson, D.: Automating commutativity analysis at the design level. In: ISSTA '04: ACM SIGSOFT international symposium on Software testing and analysis, pp. 165–174. ACM Press, New York (2004)
10. Richters, M.: A Precise Approach to Validating UML Models and OCL Constraints. PhD thesis, Universitaet Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14 (2002)
11. Bordbar, B., Anastasakis, K.: UML2Alloy: A tool for lightweight modelling of Discrete Event Systems. In: Guimarães, N., Isaías, P. (eds.) IADIS International Conference in Applied Computing 2005, vol. 1, pp. 209–216. IADIS Press, Algarve, Portugal (2005)
12. Bordbar, B., Anastasakis, K.: MDA and Analysis of Web Applications. In: Draheim, D., Weber, G. (eds.) TEAA 2005. LNCS, vol. 3888, pp. 44–55. Springer, Heidelberg (2006)
13. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture-Practice and Promise. The Addison-Wesley Object Technology Series. Addison-Wesley, Reading (2003)
14. OMG: MOF Core v. 2.0 Document Id: formal/06-01-01, `http://www.omg.org`
15. Akehurst, D.H., Bordbar, B., Evans, M.J., Howells, W.G.J., McDonald-Maier, K.D.: SiTra: Simple transformations in java. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 351–364. Springer, Heidelberg (2006)

16. OMG: UML: Superstructure. Document id: formal/05-07-04, http://www.omg.org
17. OMG: OCL Version 2.0 Document id: formal/06-05-01, http://www.omg.org
18. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: principles, techniques, and tools. Addison-Wesley, Reading (1986)
19. Wimmer, M., Kramler, G.: Bridging grammarware and modelware. In: Bruel, J.M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 159–168. Springer, Heidelberg (2006)
20. TLSWG: SSL 3.0 specification (1996), http://wp.netscape.com/eng/ssl3
21. Georg, G., Anastasakis, K., Bordbar, B., Houmb, S.H., Ray, I., Toahchoodee, M.: Verification and trade-off analysis of security properties in UML system models. Transaction. In: Software Engineering. Special Issue on Security (submitted)
22. Jackson, D.: A Comparison of Object Modelling Notations: Alloy, UML and Z (August 1999), Available at http://sdg.lcs.mit.edu/publications.html
23. Vaziri, M., Jackson, D.: Some Shortcomings of OCL, the Object Constraint Language of UML. In: Technology of Object-Oriented Languages and Systems (TOOLS 34'00), Santa Barbara, California, pp. 555–562 (2000)
24. Gogolla, M., Richters, M.: Transformation rules for UML class diagrams. In: Bézivin, J., Muller, P.-A. (eds.) The Unified Modeling Language. UML 1998: Beyond the Notation. LNCS, vol. 1618, pp. 92–106. Springer, Heidelberg (1999)
25. Wallace, C.: Using Alloy in process modelling. Information and Software Technology 45(15), 1031–1043 (2003)
26. Zito, A., Diskin, Z., Dingel, J.: Package Merge in UML 2: Practice vs. Theory? In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 185–199. Springer, Heidelberg (2006)
27. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice Hall, Upper Saddle River, NJ (1996)
28. Abrial, J.R.: The B-book: assigning programs to meanings. Cambridge University Press, New York (1996)
29. He, Y.: Comparison of the modeling languages Alloy and UML. In: Arabnia, H.R., Reza, H. (eds.) Software Engineering Research and Practice, SERP 2006, Las Vegas, Nevada, vol. 2, pp. 671–677 (2006)
30. Negri, M., Pelagatti, G., Sbattella, L.: Formal semantics of SQL queries. ACM Transactions on Database Systems (TODS) 16(3), 513–534 (1991)