SoSyM manuscript No.
(will be inserted by the editor)

On challenges of Model Transformation from UML to Alloy

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

[1] School of Computer Science, University of Birmingham, Edgbaston, Birmingham, UK
e-mail: `K.Anastasakis@cs.bham.ac.uk, B.Bordbar@cs.bham.ac.uk`

[2] Computer Science Department, Colorado State University, Fort Collins, Colorado, USA
e-mail: `georg@cs.colostate.edu, iray@cs.colostate.edu`

Received: date / Revised version: date

Abstract The Unified Modeling Language (UML) is the de facto language used in the industry for software specifications. Once an application has been formally specified, Model Driven Architecture (MDA) techniques can be applied to generate code from such specifications. Since implementing a system based on a faulty design requires additional cost and effort, it is important to analyse the UML models at earlier stages of the software development lifecycle. This paper focuses on utilizing MDA techniques to deal with the analysis of UML models and identify design faults within a specification. Specifically, we show how UML models can be automatically transformed into Alloy which, in turn, can be automatically analysed by the Alloy Analyzer. The proposed approach relies on MDA techniques to transform UML models to Alloy. This paper reports on the challenges of the model transformation from UML class diagrams and OCL to Alloy. Those issues are caused by fundamental differences in design philosophy of UML and Alloy. To facilitate better representation of Alloy concepts in the UML, the paper draws on the lessons learnt and presents a UML profile for Alloy.

Key words Alloy, Class Diagrams, MDA, OCL, UML2Alloy

1 Introduction

The Unified Modelling Language (UML) [39] is the de-facto modelling language used in the software industry for capturing requirements, design and specification of software systems. Using automated Model Driven Architecture (MDA) techniques, UML models are now widely used for the implementation of software systems by using a chain of model transformations from high-level platform independent models to lower level platform specific model and subsequently to code [30]. UML models should be analysed to ensure that faults are detected during the early stages of software development lifecycle. This, in turn, will provide significant savings in cost compared with rectifying errors after the system has been implemented. Since manual analysis is tedious and error-prone, the analysis should be automated to the extent possible [33, 29, 15, 7].

Our research utilises Alloy for the analysis of UML models consisting of class diagram and OCL. Alloy [27] is a high level modelling language for specifying Object Oriented systems. It also allows expressing first-order logic structural constrains on the model. Moreover, Alloy is supported by a software infrastructure [25], which provides fully automatic analysis of models in the form of simulation and checking the consistency of specifications. Alloy is widely used in the analysis of Object Oriented systems, among others, it has been successfully applied to the modelling and analysis of protocols in distributed systems [45], networks [18] and mission critical systems [14].

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 [27, 42]. 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. Because of these differences, model transformation from UML to Alloy has proved to be very challenging.

Our earlier paper [7] has outlined some of the challenges of the model transformation from UML class diagram and OCL to Alloy. This paper extends our previous work, by discussing further challenges related to object identifiers, multiple inheritance and supported collection constructs. The transformation rules from UML and OCL to Alloy are presented in more detail and the case study presented here is explained in more depth. Moreover this paper presents a UML profile for Alloy to allow better representation of Alloy concepts in the UML.

The paper is organised as follows. Section 2 presents on overview of basic concepts in this paper, such as the MDA and Alloy. Section 3 demonstrates our work on the definition of the transformation rules from UML to Alloy. An example of a secure e-business system, on which we have applied our approach is presented in Section 4 and Section 5 presents the transformation rules from UML and OCL to Alloy. Section 6 illustrates some of the most important issues involved in the transformation. Section 7 describes a UML profile for Alloy, developed to allow the representation of Alloy concepts in UML. An outline of our implementation and a presentation of how our method can be used to analyse the secure e-business system, can be found in Section 8. Section 9 elaborates further on some issues of the transformation and provides pointers to future work. Finally Section 10 presents approaches related to ours and Section 11 concludes the 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) [30] 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* [36]. 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 [6, 28].

Alloy: Alloy [27] 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 cannot be divided into smaller parts), *immutable* (their properties remain the same over time) and *uninterpreted* (they do not have any inherent properties) [27]. Each field belongs to a signature and represents a relation between two or more signatures. Such relations are interpreted as sets of tuples of atoms. Alloy introduces *facts* which are statements that define constraints on the elements of the model. Parameterised constraints, which are referred to as *predicates*, can be included in other predicates or facts. Alloy is supported by a fully automated constraint solver, called Alloy Analyzer [25], 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. A scope is a positive integer number, which limits the number of each model element in an instance of the system that is being analysed by the solver. 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, we refer the reader to [27, 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 Description of our Approach

This section presents a brief description of our work. An outline of our approach is depicted in Figure 1. Using the EBNF representation of the Alloy grammar [27], a MOF compliant [36] metamodel for Alloy was developed. To conduct the model transformation from UML to Alloy, a set of transformation rules has been defined, which map elements of a subset of the metamodels of class diagrams and OCL into the elements of the metamodel of Alloy. The next section presents the subset of UML that is used by our approach.

3.1 UML Class Diagrams Subset

The subset is expressive enough to represent basic class diagram concepts, such as classes, attributes, associations and OCL constraints. Our subset excludes less popular UML features, whose semantics can be expressed using classes, associations and OCL. Features not included in our UML metamodel subset, are *interfaces*, *dependencies* and *signals*.

The subset of UML used in this paper consists of the Kernel package of the UML metamodel [40, p. 22], as depicted in Figure 2. For reasons of brevity and so as not to replicate the UML specification, here we present a simplified version of the metamodel, with only the concrete metaclasses (for example we do not show the common metaclass *Element*). This metamodel will be used in Section 5, to describe the transformation rules from UML to Alloy.

3.2 Alloy Metamodel

Alloy is a textual language and its syntax is defined in terms of its EBNF [4] grammar [27, 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 [50] present a method for generating metamodels from EBNF representations. We utilised their approach to generate a MOF compliant Alloy metamodel.

Figure 3 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 subseting signatures respectively. A *SigDecl* has a signature body (*SigBody*), which can contain a sequence of constraints (*ConstraintSequence*). A signature declaration specifies zero or more 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 (*DeclSe-*

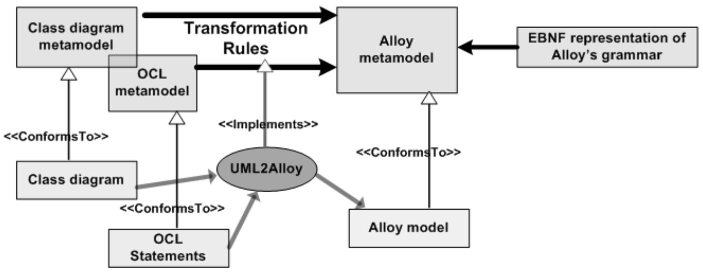

Fig. 1 Outline of our transformation method.

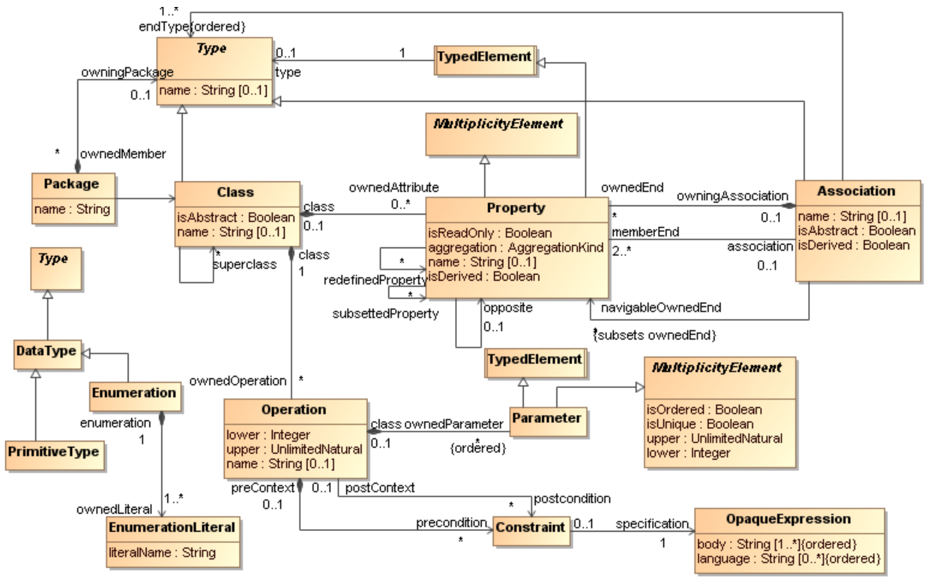

Fig. 2 A Simplified UML Metamodel

tExp) 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*.

Consulting the Alloy reference manual, we were able to devise some well-formedness rules. For example a Signature may not extend itself. This can be formally specified in OCL:

```
context ExtendsSigDecl
inv: ExtendsSigDecl.allInstances() -> forAll(
s:ExtendsSigDecl | (s <> self) implies
(s.declares.name <> self.declares.name))
```

Other essential well-formedness rules for the Alloy metamodel are included in Appendix A.

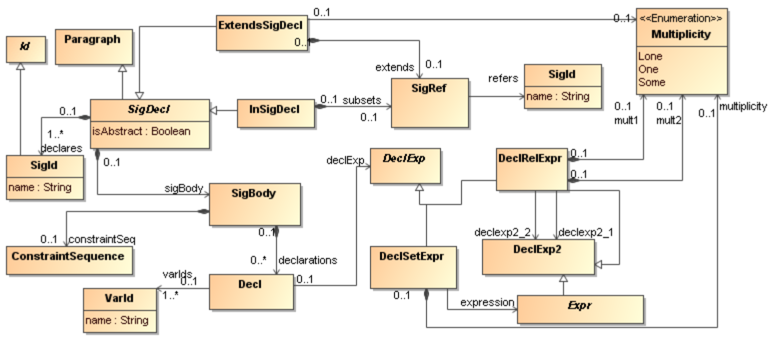

Fig. 3 A subset of the Alloy metamodel corresponding to signature declarations.

4 Example UML Class Diagram

Figure 4 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 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 [46]) authentication and confidentiality protocol. The man-in-the-middle is modelled in our example by adding an Attacker class that intercepts all communications between the Client and the e-commerce server. The Attacker may change the content of messages, exchanged between the Client and the Server. 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

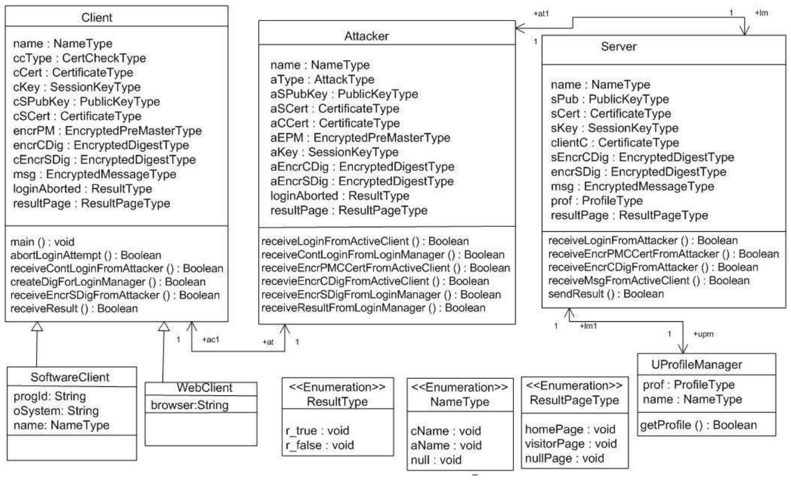
Client
name : NameType
ccType : CertCheckType
cCert : CertificateType
cKey : SessionKeyType
cSPubKey : PublicKeyType
cSCert : CertificateType
encrPM : EncryptedPreMasterType
encrCDig : EncryptedDigestType
cEncrSDig : EncryptedDigestType
msg : EncryptedMessageType
loginAborted : ResultType
resultPage : 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
+ac1
+at
1
1
+lm1
1
+upm
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. 4 A model of the SSL protocol used in login service.

encrypted, and thus confidential. If the handshake fails, all communication is aborted between Client and Server.

Figure 4 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. We have modified the original e-commerce system model to illustrate some key points in the transformation from UML to Alloy. The modifications are the addition of two specialization to the original Client class, *SoftwareClient* and *WebClient*. These specializations have different attributes, and also demonstrate different inheritance properties that become important in transforming the model to Alloy.

5 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.

5.1 Mapping Class Diagrams to Alloy

Table 1 provides the correspondence between the main elements of the UML and OCL metamodels and Alloy. In this section we present in more detail the transformation rules for the most important elements of the UML metamodel for Class diagrams, i.e. Classes and Properties. Remaining transformation rules specified in Table 1 can be explained similarly.

Table 1 Correspondence between UML and Alloy metamodel elements

UML metamodel elements	Alloy metamodel element
Package	ModuleHeader
Class	ExtendsSigDecl
Property	Decl
Multiplicity	Expr
Operation	Predicate
Parameter	Decl
Enumeration	ExntedsSigDecl
EnumerationLiteral	ExtendsSigDecl
Constraint	Expression
DataType	ExtendsSigDecl

Top level UML Classes (i.e. Classes, which are not subclasses of any other Class) are mapped to Alloy top level Signatures. The UML metaelement *Class* is mapped to the Alloy metaelement *ExtendsSigDecl*. The *name* of the Class is mapped to the *name* attribute of the *SigId* related to the *ExtendsSigDecl*. The *isAbstract* metaattribute of the Class is mapped to the *isAbstract* metaattribute of the *ExtendsSigDecl*. Top level Signatures are not related to any *SigRef*.

Subclasses (i.e. classes that extend other classes) are transformed to Alloy SubSignatures. The SubSignature is an instance of an *ExtendsSigDecl*, like top level signatures, but in this case the SubSignature is related to a *SigRef*, which references the signature it is extending.

For example, the *Client* class in the UML model of Fig. 4 is transformed to an *ExtendsSigDecl*, which *declares* a *SigId*, whose *name* is *Client*. Because *Client* is not 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.

A *Property* in UML is used to denote to either an attribute or an association end of a Class and is translated to an Alloy *field* of a signature [27, Sec. 4.2]. A UML Property is translated to an Alloy declaration of a signature (*Decl*). The *name* metaattribute of the Property is mapped to the *name* metaattribute of the *VarId* related to the *Decl*. The multiplicity of the Property is mapped to the *Multiplicity* of the *DeclSetExpr* related to the declaration (*Decl*). The type of the Property is translated to a signature reference (*SigRef*) referencing the signature, to which the class of the type of the Property, was translated.

For example, Figure 5 depicts a partial object diagram of the abstract syntax of the generated Alloy model of the *WebClient* class of Figure 4. The *WebClient* was translated to an *ExtendsSigDecl*, which extends the *Client* signature id. The *browser* attribute was transformed to a *VarId* and the type of the attribute to a *SigRef*. Using a simple MOF2Text [37] mapping we automatically translate the Alloy model abstract syntax instance of Figure 5 to the following Alloy textual notation:

```
sig WebClient extends Client{
  browser: one String}

sig String{}
```

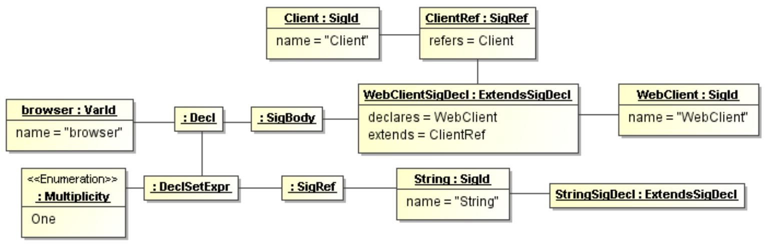

Fig. 5 A Partial Alloy Metamodel Instance of the WebClient Class of Figure 4

5.2 Mapping OCL

Most of the OCL operations on collections have a corresponding Alloy expression. The *forAll()* and *exists()* operations can be mapped to the *all* and *some* Alloy expressions respectively. Similarly the *size()* operation can be represented by the Alloy set cardinality operator (#). The *isEmpty()* and *notEmpty()* operations can be equivalently expressed using the *size()* OCL operation as suggested by Cabot and Teniente [13]. Therefore using the transformation rule for *size()*, we can express those operations in Alloy. The *includes()* and *excludes()* operations can be expressed using the Alloy set inclusion operator (*in*). Likewise OCL set operations, such as *union()*, *intersection()* and *product()* can be expressed using the equivalent Alloy set operators, +, &, and →, respectively.

Table 2 depicts the mapping between a subset of OCL and Alloy. This subset consists of the OCL statements that can be directly mapped to Alloy. For instance, the syntax of the OCL *forAll()* operation is shown in the first cell of the table. The *forAll()* operation is applied on a collection (*col*), defines a variable (v) of type *Type* and a boolean expression (*be*). This is transformed to an *all* Alloy expression. The variable v is mapped to the equivalent Alloy variable expression (*TR(v)*), the collection is transformed to the equivalent Alloy expression (*TR(col)*) and finally the boolean expression is translated to the equivalent Alloy boolean expression (*TR(be)*).

There are however a number of OCL constructs that cannot be expressed in Alloy. Alloy has a very simple type system and does not support

Table 2 Correspondence between OCL and Alloy expressions

OCL expression	**Alloy expression**
col → forAll(v:Type | be)	all TR(v):TR(col) | TR(be)
col → exists(v:Type | be)	some TR(v):TR(col) | TR(be)
expr1 and expr2	TR(expr1) && TR(exp2)
expr1 or expr2	TR(expr1) || TR(expr2)
not expr	! TR(expr)
col → size()	#TR(col)
col → includes(o:T)	TR(o) in TR(col)
col → excludes(o:T)	TR(o) !in TR(col)
col1 → includesAll(col2)	TR(col2) in TR(col1)
col1 → excludesAll(col2)	TR(col2) !in TR(col1)
col1 → including(o:T)	TR(col1) + TR(o)
col1 → excluding(o:T)	TR(col1) - TR(o)
col → isEmpty()	#TR(col) = 0
col → notEmpty()	#TR(col) != 0
expr.PropertyCallExpr	TR(expr).TR(PropertyCallExpr)
if cond then expr1 else expr2	TR(cond) ⇒ TR(expr1) else TR(expr2)
expr.oclIsUndefined	#TR(expr) = 0
expr → oclIsKindOf(o:T)	TR(o) in TR(expr)
col1 → union(col2)	TR(col1) + TR(col2)
col1 → intersection(col2)	TR(col1) & TR(col2)
col1 → product(col2)	TR(col1) → TR(col2)
col → sum()	sum TR(col)
col1 → symmetricDifference(col2)	(TR(col1) + TR(col2)) - (TR(col1) & TR(col2))

attributes overriding. As a result OCL's casting operations are not required. Additionally as a purely declarative language, Alloy does not support expressions with imperative flavour such as OCL's *iterate* operation. OCL statements that do not appear in Table 2 cannot be generally translated to Alloy. OCL constraints that cannot be mapped to Alloy are not handled by our model transformation. The modeller needs to be able to express the constraints, using the OCL subset depicted in Table 2.

6 Differences between UML and Alloy which Influence the Model 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, including inheritance, overriding and predefined types [26]. 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.

6.1 Object Ids vs Atoms:

Since each Class is mapped to an Alloy Signature, the instances of a Class, will map to an instance of a Signature. In UML a Class denotes a set of object identifiers (*Object Ids*) [38, Ap. A.1.2.1]. In Alloy a Signature denotes to a set of atoms. Atoms are *indivisible* (they cannot be divided into smaller parts), *immutable* (their properties remain the same over time) and *uninterpreted* (they do not have any inherent properties) [27]. An *Object Id* in UML is used to uniquely identify an instance of a Class, in the same way that an *atom* in Alloy, identifies an instance of a Signature. The notion of Object Id maps conceptually to the notion of an atom when dealing with static systems. However, considering the notion of Object Ids conceptually equal to the notion of atoms, has certain implications when modelling dynamic systems, as described in Section 6.9.

6.2 Redefinition:

A UML *Property* or *Operation* of a subclass can redefine one or more *Properties* or *Operations* of one or more of its superclasses. According to the UML standard, a redefining property can define *additional* constraints on the redefined property [40, Sec. 7.3.46]. A redefining Property has usually the same name as the redefined Property. The notion of redefinition corresponds to the well known concept of operation and attribute overriding in Object-Oriented programming.

Alloy does not directly support the notion of redefinition. More specifically signatures that belong to the same hierarchy may not define fields with the same name. This is also the stance the UML formal semantics takes on this issue [38, p. 182]. Our approach follows the UML formal semantics view on the issue and as discussed in Section 7, we do not allow UML class diagrams with redefinition in our approach. As a result a diagram such as that of Figure 4, cannot be directly represented in Alloy, since the *SoftwareClient* has an attribute (*name*), which redefines the attribute of the *Client* class.

6.3 Multiple Inheritance:

A UML class can extend more than one superclasses. For example, in Figure 6 an instance of a *CameraPhone* is an instance of both a *Phone* and a *Camera* at the same time [42, Sec. 3.4.1]. More formally: $I(\text{CameraPhone}) = I(\text{Phone}) \cap I(\text{Camera})$, where $I(\text{CameraPhone})$, $I(\text{Phone})$ and $I(\text{Camera})$ denote to the set of *Object Ids* of the instances of the CameraPhone, Phone and Camera classes respectively.

Using our transformation rules, the diagram of Figure 6 will be transformed to an Alloy model with three signatures, a *Phone*, a *Camera* and a *CameraPhone* signature. By definition top-level Alloy signatures (like *Camera* and *Phone*) define disjoint sets. More formally: $I(\text{Phone}) \cap I(\text{Camera}) = \emptyset$.

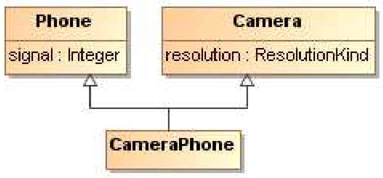

Fig. 6 Multiple Inheritance

It is therefore evident that in this case: $I(\text{CameraPhone}) = \emptyset$. In general, multiple inheritance cannot be directly represented in Alloy. As we will demonstrate in Section 7, we do not allow UML class diagrams with multiple inheritance to be defined in our method.

In some specific cases, multiple inheritance can be represented in Alloy, as demonstrated by Jackson [27, p. 94]. More specifically, if the classes participating in the multiple inheritance have a common superclass, they can be expressed in Alloy. However, this case of multiple inheritance is not generic enough to be incorporated in our method.

6.4 Collections

UML has inherent support for a number of collection constructs. In particular the OCL standard defines *Sets*, *Bags*, *OrderedSets* and *Sequences*) [38, Sec. 7.5.11]. On the other hand Alloy only supports *Sets*. Recently sequences have been introduced to the Alloy language. Here we show that it is possible to express the UML concepts of Bags and OrderedSets using the notion of Sequences in Alloy. An Alloy Sequence is defined as a relation between an integer value and a user defined signature. The integer value denotes to the index of the element in the Sequence.

In UML a *Bag* is collection like a Set, but allows elements to appear more than once in the collection. A Bag can be represented as an Alloy Sequence, ignoring the integer value, which represents the index of the element in the Sequence. Similarly a UML *OrderedSet* is a *Set*, whose elements are ordered. Again this can be easily expressed using an Alloy Sequence, with an additional constraint that no two distinct elements in the Sequence can be the same.

Nested Collections: The UML allows for nested collection (i.e. Collections of Collection) [38, Sec. 7.5.12]. In Alloy on the other hand all collections are flat and it is not possible to express higher-order relations [27, p. 41]. As a result if a collection of a collection is defined in OCL, it is rejected by our transformation.

6.5 Namespace:

All UML model elements are defined in a *namespace* [38, 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 [27, 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*' [38, p. 178]. In Alloy on the other hand signature names, have to be distinct from their field names.

This difference between the two languages is tackled in our UML profile for Alloy described in Section 7. For example our profile does not allow attributes names and class names to be the same. This ensures that no naming conflicts appear in the generated Alloy model.

6.6 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 [27, 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 [38, p. 184].

These differences in the two languages stem from the fact that UML and Alloy have different design philosophies. For example, 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 [48].

To explain this consider the navigation dot (.). In Alloy it is treated as the relational join [27, 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, as in UML [38, Ap. A.2.1.1]. Let us assume in the model of Fig. 4, the multiplicity of the *at* role is 0..1 (i.e. a Client can be related to 0 or 1 Attackers). Now, let us consider the following OCL fragment: '*self.at*'.

In UML if the instance of the Client in which this OCL invariant is evaluated is related to no Attacker, the expression '*self.at*' will denote to an *undefined* value. In an equivalent Alloy model, such an expression evaluates to an empty set.

In order to transform OCL's three valued logic into Alloy's two valued logic, we make a counter intuitive, but necessary assumption. That the empty set evaluates to the same as the *undefined* value. This is due to OCL's implicit conversion of scalars to sets. In particular according to the OCL specification [38, p. 81], if a collection operation is applied on a scalar, the scalar is automatically converted to a set. Assume the following two statements in the context of the Client, in our example shown in Figure 4:

```
inv collSize: self.at -> size() = 0
inv undef: self.at.oclIsUndefined()
```

Both *collSize* and *undef* invariants evaluate to *true* if no Attacker is related to the Client. Considering the empty set equal to the undefined value, is an important assumption that allows us to translate OCL statements to Alloy. Such an assumption has also been made by Akehurst et al. [5] in their translation of UML into Java 5.

In general the *undefined* value in OCL is used to denote the absence of a value or a run time error [38, Ap. A]. If it is important to explicitly model the absence of a value from a model element, it is possible to do so on the model level. For example, the *NameType* enumeration type of Figure 4 defines an enumeration literal called *null*. This literal represents the absence of a value in an attribute of type *NameType* in the model and can be used in OCL in the following way:

```
context Client
inv: (self.name <> NameType::null) implies
    (expression with the name attribute not null)
```

6.7 Aggregation and Composition:

In UML special kinds of binary association exist to denote a Whole-Part Relationship (WPR) [8] between classes. More specifically the UML provides the notions of aggregation (shared aggregation) to denote weak ownership and composition (composite aggregation) to denote strong ownership. The exact meaning of aggregation and its difference from composition has received considerable attention. Fowler refers to aggregation as '*one of the most frequent sources of confusion*' [17, p. 67] and Rumbaugh et al. suggest to '*think of it as a modeling placebo*' [43, p. 148], while Henderson-Sellers and Barbier ask: '*what is this thing called aggregation?*' [24]. Moreover the UML standard specifies that: '*Precise semantics of shared aggregation varies by application area and modeler*' [40, Sec. 7.3.2].

It is therefore evident that there is not a clear view on the *exact* semantics of aggregation and composition amongst the UML community. However, despite the different interpretations, there are some primary properties of aggregation and composition that most researchers agree on [20, 43]. More specifically the weak ownership semantics of aggregation do not allow self references [1]. The strong ownership semantics of composition, in addition to not allowing self references, impose that in an WPR, the part can exist only if an instance of the whole exists and two wholes cannot share the same part(s).

[1] In a Whole-Part Relationship, the whole cannot be part of itself.

These rules have been formalised by Gogolla and Richters [20]. The authors present a methodical way of refactoring aggregations or compositions as standard binary associations enriched with OCL constraints to depict the additional semantics of aggregation and composition. This methodology is well suited for our transformation to Alloy, since we have already defined the transformation rules for binary associations and a subset of OCL. As a result we have not explicitly defined transformation rules for aggregation and composition. We require that they are expressed as standard binary association and OCL is used to capture the additional semantics.

6.8 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. 4 is of type String.

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

Therefore, 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 requires that all primitive types, apart from Integers, are defined by the modeller on the model level as UML Datatypes.

6.9 Static vs Dynamic Models:

Models in Alloy are static, i.e. they capture the entities of a system, relationship between the entities and constraints that the system must satisfy. An Alloy model defines a single instance of a system, on which the constraints are satisfied. In particular, Alloy models do not have an inherent notion of *states*, or any form of built in notion of statemachines [27, 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 [38, p. 185].

Since UML has an implicit notion of states, while Alloy does not support it directly, additional complexity arises in the transformation. To explain this, 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 [38, 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 cannot be true (i.e. resultPage will either be the *nullPage* or *homepage*, but not both at the same time).

Alloy has been successfully applied to the analysis of dynamic systems [14, 45, 49]. The analysis of dynamics in Alloy is carried out by modelling explicitly the notion of *state* on the model level. To model the notion of *state* in Alloy, a number of patterns have been proposed in the literature. The most popular are the *global state* [3] and *tick based* modelling [45]. Elaborating further on how a state can be represented in Alloy is out of the scope of this work.

The solution we propose to this issue, is similar to existing Alloy approaches, which model the state on the model level. More specifically as demonstrated in [10], we require a class that represent the states, to be modelled on the class diagram. The *dynamic* stereotype presented in Section 7.2 is used to annotate the association ends whose values change over time. This is done, so that the modeller does not need to specify frame conditions [11] for all class properties, but only for those stereotyped, whose value may change from state to state (i.e. properties stereotyped as *dynamic*).

In the model of Figure 4 we do not need to use the notion of *state* on the model level, because we are only interested in the messages exchanged between the Client and the Server, but not the order of the messages exchanged. Consequently no more than one *state* is required to analyse it.

To address the issues presented in this section, we have developed a UML profile for Alloy, which is presented in the next section.

7 UML Profile for Alloy

This section presents a UML profile for Alloy. There are two main reasons for defining such a profile. Firstly it defines additional constraints on the

UML metamodel elements, in order to prevent UML models which cannot be translated to Alloy. As discussed previously, Alloy does not directly support UML notions, such as multiple inheritance. Consequently a class diagram created on the basis of the profile cannot include multiple inheritance. The second purpose of this profile is to define a number of stereotypes, which can be used by the modeller to express Alloy concepts, such as the *scope* [2] in UML. As discussed later on, the stereotypes provide the ability to automatically analyse UML class diagrams via Alloy.

7.1 Profile Constraints

Figure 2 shows a simplified version of the UML Kernel package [40]. This metamodel allows the creation of UML class diagrams that cannot be represented in Alloy. More specifically it is possible to develop a UML class diagram with attributes redefinition. In order to overcome this problem, we have extended the UML metamodel elements, with additional constraints that forbid the definition of UML concepts, which are not representable in Alloy. For example a UML *NamedElement* may or may not have a name [40, Sec. 7.3.3]. Our profile requires that all elements have a (unique) name for identification. The following OCL statement is used by our profile to depict this constraint:

```
context Class
inv: self.name -> size() = 1 and self.name <> ""
```

An important extension of UML by the profile is that by default Properties are *readonly*. Once the value of a Property is set, it cannot be changed. This is because the fact that the equivalent notion of *Properties* in Alloy (i.e. *fields*) are immutable, as discussed in Section 6.9.

Such supplementary constraints, which are presented in Appendix B, were imposed on the UML metamodel to allow for the automated translation of a UML class diagram to Alloy. In addition to those constraints we have also defined a number of stereotypes that can be used to express Alloy concepts in UML.

7.2 Stereotypes

As discussed in Section 2, the Alloy language has notions such as the *scope*, *simulation* and *assertion* commands, which allow it to perform fully automated analysis of models. On the other hand the UML does not have such concepts. Since our work aims to make UML class diagrams fully analysable,

[2] The term *scope* here is used in the context of Alloy [27, Sec. 5.1.2] (i.e. a scope is a number that denotes to the maximum number of model elements the Alloy Analyzer will use for the analysis).

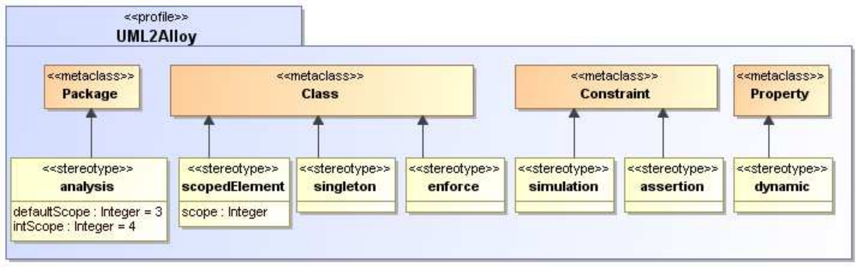

Fig. 7 The Stereotypes defined by our profile

using Alloy, we need to extend the UML class diagram notation to introduce concepts such as the *scope*, *simulation* and *assertion* commands. This is achieved with the help of the stereotypes presented in this section.

Figure 7 depicts the stereotypes defined in our UML profile for Alloy. The *analysis* stereotype is used on Packages, *scopedElement*, *singleton* and *enforce* is used on Classes and the *assertion* and *simulation* stereotypes are used on Constraints. Finally the *dynamic* stereotype can be used on Properties, to change the the default *readonly* metaatribute. In the following we briefly introduce the stereotypes and their use.

analysis: This stereotype is used on UML Packages that are going to be analysed using our method. A UML class diagram is required to have *exactly one* Package stereotyped as *analysis*. The *analysis* stereotype defines two attributes (also called *tagged values*), the *defaultScope* and *intScope*. These tagged values are used during the transformation to set the default Alloy scope and the scope for integer numbers respectively. A *defaultScope* and *intScope* are positive integer numbers. Following Alloy's approach, if no *defaultScope* tagged value has been defined in the model, the default is 3. Similarly if no default tagged value has been defined in the model for integer values, the default scope is 4.

scopedElement: Each class in a class diagram can be stereotyped as a *scopedElement*. The scopedElement stereotype defines a tagged value (*scope*), which is used to limit the number of instances of an element when a system instance is being checked by the SAT solver. This is used to override the *defaultScope* attribute of the *analysis* stereotype in order to define a different scope for the particular class on which the stereotype is applied.

singleton: A *singleton* stereotype can be applied to a Class. Classes annotated with this stereotype can only have *exactly* one instance in the model.

enforce: In general an instance of a UML class diagram may be partial (i.e. some classes may not have any instances). This stereotype is used on classes that are required to have *at least one* instance during the analysis.

simulation: In Alloy, a first-order logic statement can be used to simulate a model. This statement corresponds to the Alloy *run* command [27, Section 4.6]. Similarly in a UML class diagram an OCL constraint can be used to simulate the model. We use the *simulation* stereotype for this purpose. More specifically an OCL statement, which is stereotyped as simulation, will be automatically translated to an Alloy simulation (*run*) command. An Alloy run command can be used with the Alloy Analyzer to create a random instance of the model that conforms to the statement and the constraints of the model.

assertion: Similarly to simulation commands, *assertion* commands can be used to check if a statement that depicts a property of the system, is satisfied by the model. In a class diagram, an OCL statement can be used to depict an assertion. The *assertion* stereotype can be used on OCL statements, which will be translated to Alloy assertions.

dynamic: As we discussed earlier in Section 7 our UML profile for Alloy enforces that by default *Properties* (i.e. Attributes and Association Ends) are readonly. However, often values of properties change over time. Such properties need to be stereotyped as *dynamic*.

8 Implementation and Analysis

This section presents a brief overview of our tool that implements the rules presented in Section 5. This sections ends with a description of the outcome of the analysis of the SSL protocol presented in Section 4.

8.1 Implementation

The transformation rules presented in the previous section, have been implemented in a tool called UML2Alloy. The tool uses the Kent Modelling Framework (KMF) [1] XMI reader and OCL parser. The transformation rules from UML to Alloy are implemented in the SiTra [6] model transformation engine.

UML2Alloy parses an XMI file with the UML class diagram and OCL constraints and generates an Alloy model. The tool interacts with the Alloy Analyzer API to automatically analyse the generated Alloy model. Currently the results of the analysis are represented using the Alloy Analyzer output. In the future we plan to represent the result of the analysis using UML object diagrams.

The current version of the tool requires human intervention to carry out the translation (for example to set the scope of the model elements), however in a future release of the tool, we will provide support for the UML Profile for Alloy presented in Section 7, which will allow fully automated analysis of UML models compliant with the profile.

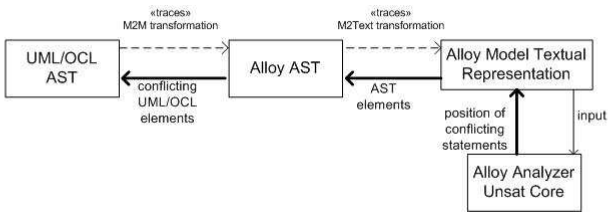
«traces»
M2M transformation
«traces»
M2Text transformation
UML/OCL AST
Alloy AST
Alloy Model Textual Representation
conflicting UML/OCL elements
AST elements
position of conflicting statements
input
Alloy Analyzer Unsat Core

Fig. 8 Scenario of Traceability of the Unsat Core

One of the most powerful features of the Alloy Analyzer, is the *Unsat Core* [44] facility. In the case of overconstrained models (i.e. models for which the Analyzer cannot provide an instance, due to conflicting constraints), the Unsat Core functionality can locate the conflicting statements that cause the inconsistency. In order to incorporate this feature in our work, we had to extend SiTra with Model to Model (*M2M*) and Model to Text (*M2Text*) tracing.

Figure 8 illustrates a scenario of how the Alloy's Unsat Core functionality can be used in our UML2Alloy transformation framework. Initially the UML/OCL model is transformed to an instance of the Alloy metamodel (*Alloy AST*), defined in Section 3.2. This is a Model to Model (*M2M*) transformation. The instance of the Alloy metamodel is then transformed to the Alloy textual notation, through a Model to Text (*M2Text*) transformation. The Alloy Analyzer API is then used to analyse the Alloy model. If the Analyzer cannot produce an instance, it returns the positions of the conflicting statements in the Alloy textual representation. Using our M2Text tracing, we can trace back those positions to the Alloy AST elements responsible for the inconsistency. Finally using the M2M tracing we can trace back the original UML/OCL model elements responsible for the conflict.

The next two subsections present how UML2Alloy can be used to analyse the e-business system presented in Section 4.

8.2 Description of the Secure System

We applied our model transformation rules from UML to Alloy on the example model presented in Sect. 4 for the man-in-the-middle attack. A man-in-the-middle attack allows an attacker to intercept and use information that may be confidential. We consider two variations of this type of attack. The first is an active attack - all messages flow through the attacker and not through a direct association between a requestor (e.g. Client) and authenticator (e.g. Server). The attacker poses as the authenticator from the point of view of the requestor, and as the requestor from the point of view of the authenticator. The attacker relays messages between the requestor and the authenticator until private information has been obtained. The attacker can change information in messages, and messages can be inserted or deleted into the communication flow.

In the second variation, the attacker does not change information in messages, insert or delete messages in the communication, but rather eavesdrops on the message flow between the requestor and the authenticator. This kind of attack is a passive attack. The information obtained can be used later to impersonate the requestor, or to access confidential information.

In order for the attacker to obtain user information, the attacker can eavesdrop until this information passes by (i.e. in our example, a homePage containing sensitive user information is returned to the Client). This represents a passive attack on the login service, and encryption schemes such as that used by SSL are enough to prevent it. The attacker can also impersonate the Client, changing the information flowing to the Server in hopes of tricking the Server to reveal confidential information, such as a session key that will then allow the attacker to intercept and decode confidential messages. In this active attack case, authentication of the message sender then becomes important so that Server can trust that a message containing, for example, SSL protocol information, actually comes from the Client.

In an active attack against an SSL-protected Server, the following steps occur:

1. The attacker substitutes its own certificate in the first message from the Server to the Client.
2. If the Client is not performing a full server certificate validation (i.e. the IP address in the certificate is not checked against the Server IP address, or the attacker has managed to create a certificate with the Server IP address that still appears to be signed by a trusted certificate authority), it will use the attacker public key instead of the Server public key to encrypt the next message.
3. The attacker can decrypt this message using its private key, then re-encrypt the message using the Server's public key. Once the attacker has access to this message from the Client, it can create session encryption keys and other SSL secrets just like the Client and the Server.
4. The attacker must keep track of all handshake messages sent by the Server and the Client, then intercept and change digest messages created from these handshake messages, as part of the SSL protocol. Please see the SSL specification for a complete description of the SSL protocol [46].

8.3 Analysis via UML2Alloy and Results

The original version of the UML model of Figure 4 (i.e. without the *SoftwareClient* and *WebClient* classes, which were added to emphasise the differences between UML and Alloy) was translated to Alloy, using UML2Alloy. 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) }
```

We checked this assertion for the case of the active attack described in the previous section (i.e. when the Attacker changes information in the messages exchanged between the Client and the Server). The analysis was carried out for a scope of six. This means that the analyser probed to find an instance of the model that violates the assertion using up to six instances of each model element. In the case of an Active attacker the Alloy Analyzer produced a counterexample, i.e. an instance of the model where the Attacker had possession of the symmetric key and thus access to the information exchanged between the Client and the Server.

Step number 2 above points out the problem that an active attack can cause. If the Client does not verify the certificate from the Server completely, it can be fooled into using the Attacker public key to encrypt a critical message. This message normally allows the Client and Server to independently compute session and digest keys. Allowing the Attacker access to this message allows the Attacker to also independently compute these keys.

The same assertion was also checked for the case of a passive attack for a scope [27, 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. 4 (for example, Client, Attacker, Server). The assertion produced no counterexample for the case of the passive attack (i.e. when the Attacker only relays information between the Client and the Server).

9 Discussion

This section presents a discussion on further details of our approach and suggests directions for future work.

A difference between UML and Alloy, is that while the former has two concepts to depict relations between model elements (i.e. Association and Properties) the latter has only one concept (i.e. Fields). In Section 5.1 we explained that we translate UML Properties to Alloy Fields. In fact the notion of a UML Association, maps more precisely to the notion of an

Alloy Field. A UML Association denotes to a set of tuples whose values refer to typed instances [40, Sec. 7.3.3]. Similarly an Alloy Field denotes to a set of tuples, whose values refer to the atoms of the Signatures of the Field declaration.

The reason for mapping a UML Property to an Alloy Field is explained if we take into account OCL. OCL uses the Property name as a path reference. Consequently our UML Property to Alloy Field transformation allows us to translate OCL to Alloy without additional complications.

Another interesting remark is the expressiveness of our approach, depicted in Figure 9. As discussed previously, OCL has a number of notions (such as the *iterate* and casting operations), which cannot be expressed in Alloy and are thus not supported by our work. On the other hand Alloy's relational logic, provides a number of operators to directly manipulate relations (for example it is easy to express the transitive closure or the transpose of a binary relation [27]). OCL does not directly support such operations. Therefore it is not possible to take advantage of the full expressive power of Alloy, without extending OCL. Consequently the expressiveness of the expressions supported by our work is the intersection of the expressions supported by OCL and Alloy.

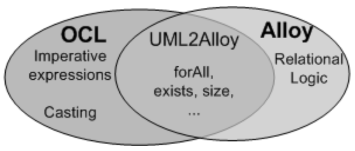

Fig. 9 Expressiveness of Alloy, OCL and UML2Alloy

10 Related Work

Formalising UML for the purpose of analysis is a popular approach. Evans et al. [15] propose the use of Z [51] as the underlying semantics for UML. Marcano and Levy [33] advocate the use of B [2], while Kim [29] makes use of an MDA method to translate a subset of UML to Object-Z. Unlike our method, which is based on Alloy's ability for fully automated analysis, these methods rely on theorem provers to carry out the analysis, which requires human intervention and special experience that complicate the process of analysis.

A number of UML tools also provides support for analysis. For example, the USE tool (UML Specification Environment) [42] is a powerful instance evaluator with the ability of simulation. More specifically it is possible to use the USE tool to generate snapshots that conform to the model. It is also possible to check if a specific instance of the model conforms to the constraints. This method requires that the instances to be checked are generated manually. To overcome this issue Gogolla et al. [21] suggest a scripting language that automates the process of generating instances. This method can be potentially used to automatically check a large number of instances against the model. In contrast, our approach uses the Alloy Analyzer, which automatically searches the state space exhaustively (up to the user specified scope), resulting in a higher degree of confidence. However, unlike our approach USE provides support for UML concepts, which are not expressible by Alloy and therefore not supported by our method. For example USE has inherent support for OCL's three valued logic and multiple inheritance.

Another category of UML tools rely on theorem provers for conducting the analysis. The KeY tool [9] formalizes OCL with the help of dynamic logic [22] and provides an interactive theorem prover environment for the analysis of UML models and their implementation. HOL-OCL [12] is another tool that transforms OCL to Higher Order Logic (HOL) formulas that can be analysed by the Isabelle [35] theorem prover. All these methods require guidance and special expertise to operate the theorem prover environment. Most application developers lack such expertise. On the other hand, our method relies on SAT-solvers and as a result the analysis if fully automated.

Using Alloy to formalise UML has also received considerable attention. More specifically Dennis et al. [14] use Alloy to expose hidden flaws in the UML design of a radiation therapy machine. Georg et al. [18] 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.

A number of secure systems have also been modelled and analysed directly in Alloy. Torlak et al. [47] have used Alloy to analyse man-in-the-middle attacks. They introduce the idea of *knowledge flow*, which models how knowledge (information) is exchanged between the participants of a protocol handshake. In particular, like our method, they focus on the information that the man-in-the-middle can possess, irrespective of the order the messages are exchanged between the participants. The protocol under investigation was modelled directly in the Alloy language and analysed. They used their method on the Needham and Schröeder [34] protocol and verified the existence of a flow already discovered by Lowe [32]. Their method was developed to prove certain properties of security protocols, thus they formally prove the completeness and soundness of their method. It is not possible to easily follow such an approach, without knowledge of formal methods. On the other hand our study presented in Section 8 is used as a proof of concept for our UML2Alloy transformation and it is using concepts such as UML class diagrams and OCL, which are more familiar to the average software developer. Another study, which has successfully used Alloy to model secure systems, was conducted by Ramananandro [41]. The author exposed defects in the Z specification of a smart card system, by translating it to Alloy. Such studies demonstrate the suitability of Alloy as a language for the analysis of models.

Finally there have been studies on the comparison of languages of UML and Alloy [26, 23]. However, they do not use model driven approaches to demonstrate the differences.

11 Conclusions and Future Work

This paper deals with the analysis of UML models captured as class diagrams, enriched with OCL statements, modelling various constraints on the system. Using a model driven approach, UML models are automatically transformed to corresponding Alloy representations. Alloy models can then be analysed automatically, with the help of Alloy Analyzer. The emphasis of this paper is on the underlying model transformation which maps UML to Alloy.

The paper outlines differences between UML and Alloy, which influence the transformation between the two languages. Such differences stem from the different approaches adopted by UML and Alloy on fundamental object oriented notions such as identifiers, inheritance, type system, partial functions and handling of dynamic behaviour in models.

In this work, we also present a UML profile for Alloy. The profile imports elements of the UML class diagrams metamodel and applied additional constraints on them. This prevents instantiation of UML models, which can not be automatically translated to Alloy, using our work. Additionally the profile defines a number of stereotypes, which are used to represent Alloy concepts, such as the *scope* and *simulation* commands in UML. [illegible] development of the profile allows the fully automated analysis of class diagrams, via Alloy.

A number of issues remain open for future research. ~~In this paper we have not presented any transformation~~ rules for association classes, n-ary and qualified associations. ~~Observing the~~ UML profile in Appendix ~~B, it is clear that it is not possible to~~ express a class diagram with association classes or n-ary/qualified associations. ~~Our approach requires that those constructs, need to be~~ refactored to binary associations with OCL to represent the additional semantics.

Fowler [17] presents a method called *class promotion*, to refactor association classes to binary associations with OCL. Akehurst et al. [5] present an approach on how to translate qualified associations to standard binary associations. In particular they replace a qualified association, with a class that explicitly represents the qualified association semantics. Finally the approach described by Gogolla and Richters [19] can be employed to refactor n-ary associations to binary ones. Translating association classes, qualified and n-ary associations to Alloy remains for future research.

Another interesting direction for further research is to apply emerging model transformation testing techniques [16, 31] on the method presented in this paper. For example, it is important to test whether the UML to Alloy transformation rules can generate a syntactically incorrect Alloy model.

Applying such model transformation testing techniques can test if the model transformation implementation satisfies certain criteria, such as coverage, termination and syntactic correctness [31].

References

1. Kent Modelling Framework. `http://www.cs.kent.ac.uk/projects/kmf`. School of Computing, University of Kent.
2. J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996. ISBN 0-521-49619-5.
3. Aditya Agrawal. Graph Rewriting and Transformation (GReAT): A Solution For The Model Integrated Computing (MIC) Bottleneck. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 364–368, Montreal, Canada, 2003. IEEE Computer Society. URL `http://csdl.computer.org/comp/proceedings/ase/2003/2035/00/20350364abs.htm`.
4. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison Wesley, Reading, MA, 1986.
5. D. Akehurst, G. Howells, and K. Mcdonald-Maier. Implementing associations: UML 2.0 to Java 5. *Software and Systems Modeling*, 6(1): 3–35, March 2007. doi: 10.1007/s10270-006-0020-1.
6. David H. Akehurst, Behzad Bordbar, M. J. Evans, W. G. J. Howells, and Klaus D. McDonald-Maier. SiTra: Simple transformations in java. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006*, volume 4199 of *Lecture Notes in Computer Science*, pages 351–364, Genova, Italy, 2006. Springer.
7. Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. UML2Alloy: A Challenging Model Transformation. In G. Engels, B. Opdyke, D.C. Schmidt, and F. Weil, editors, *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems*, volume 4735 of *LNCS*, pages 436–450, Nashville, USA, 2007. Springer.
8. Alessandro Artale, Enrico Franconi, Nicola Guarino, and Luca Pazzi. Part-whole relations in object-centered systems: An overview. *Data & Knowledge Engineering*, 20(3):347–383, 1996.
9. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software. The KeY Approach*. Springer-Verlag New York, Secaucus, NJ, USA, 2007. ISBN 354068977X.
10. Behzad Bordbar and Kyriakos Anastasakis. UML2Alloy: A tool for lightweight modelling of Discrete Event Systems. In Nuno Guimarães and Pedro Isaías, editors, *IADIS International Conference in Applied Computing 2005*, volume 1, pages 209–216, Algarve, Portugal, February 2005. IADIS Press. ISBN 972-99353-6-X.
11. Alex Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *IEEE Trans. Softw. Eng.*, 21(10):

785–798, 1995. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.469460.

12. Achim D. Brucker. *An Interactive Proof Environment for Object-oriented Specifications.* Ph.d. thesis, ETH Zurich, March 2007. URL `http://www.brucker.ch/bibliography/abstract/brucker-interactive-2007`. ETH Dissertation No. 17097.
13. J. Cabot and E. Teniente. Transformation techniques for ocl constraints. *Sci. Comput. Program.*, 68(3):152–168, 2007. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2007.05.001.
14. Greg Dennis, Robert Seater, Derek Rayside, and Daniel Jackson. Automating commutativity analysis at the design level. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 165–174. ACM Press, 2004. ISBN 1-58113-820-2. doi: http://doi.acm.org/10.1145/1007512.1007535.
15. Andy Evans, Robert France, and Emanuel Grant. Towards Formal Reasoning with UML Models. In *Proceedings of the OOPSLA'99 Workshop on Behavioral Semantics*, 1999.
16. F. Fleurey, J. Steel, and B. Baudry. Validation in model-driven engineering: testing model transformations. In *First International Workshop on Model, Design and Validation*, pages 29– 40, 2004.
17. Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language.* Object Technology Series. Addison Wesley, 3rd edition, 2003. ISBN 0321193687.
18. Geri Georg, James Bieman, and Robert B. France. Using Alloy and UML/OCL to Specify Run-Time Configuration Management: A Case Study. In Andy Evans, Robert France, Ana Moreira, and Bernhard Rumpe, editors, *Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists.*, volume P-7 of *LNI*, pages 128–141. German Informatics Society, 2001.
19. Martin Gogolla and Mark Richters. Expressing uml class diagrams properties with ocl. In *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, pages 85–114, London, UK, 2002. Springer-Verlag. ISBN 3-540-43169-1.
20. Martin Gogolla and Mark Richters. Transformation rules for UML class diagrams. In *"UML"': '98: Selected papers from the First International Workshop on The Unified Modeling Language "UML": '98*, pages 92–106, London, UK, 1999. Springer-Verlag. ISBN 3-540-66252-9.
21. Martin Gogolla, Jörn Bohling, and Mark Richters. Validating UML and OCL Models in USE by Automatic Snapshot Generation. *Journal on Software and System Modeling*, 4(4):386–398, 2005.
22. D Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic.* MIT Press, 2000.
23. Yujing He. Comparison of the modeling languages Alloy and UML. In Hamid R. Arabnia and Hassan Reza, editors, *Software Engineering Research and Practice, SERP 2006*, volume 2, pages 671–677, Las Vegas, Nevada, USA, 2006.

24. B. Henderson-Sellers and F. Barbier. What is this thing called aggregation? In *TOOLS '99: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 236, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0275-X.
25. Daniel Jackson. Alloy Analyzer website. `http://alloy.mit.edu/`.
26. Daniel Jackson. A Comparison of Object Modelling Notations: Alloy, UML and Z. Available at: http://sdg.lcs.mit.edu/publications.html, August 1999.
27. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, London, England, 2006.
28. Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *LNCS*, pages 128–138. Springer, 2006. URL `http://www.lina.sciences.univ-nantes.fr/Publications/2006/JK06a`.
29. Soon-Kyeong Kim. *A Metamodel-based Approach to Integrate Object-Oriented Graphical and Formal Specification Techniques.* PhD thesis, University of Queensland, Brisbane, Australia, 2002.
30. Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture–Practice and Promise.* The Addison-Wesley Object Technology Series. Addison-Wesley, 2003. ISBN 032119442X.
31. Jochen Malte Küster and Mohamed Abd-El-Razik. Validation of model transformations - first experiences using a white box approach. In Thomas Kühne, editor, *MoDELS Workshops*, volume 4364 of *Lecture Notes in Computer Science*, pages 193–204. Springer, 2006. ISBN 978-3-540-69488-5.
32. Gavin Lowe. Breaking and fixing the Needham-Schröeder public-key protocol using FDR. In *TACAs '96: Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 147–166, London, UK, 1996. Springer-Verlag. ISBN 3-540-61042-1.
33. R. Marcano and N. Levy. Using B formal specifications for analysis and verification of UML/OCL models. In Ludwik Kuzniarz, Gianna Reggio, Jean Louis Sourrouille, and Zbigniew Huzar, editors, *Blekinge Institute of Technology, Research Report 2002:06.*, pages 91–105. Department of Software Engineering and Computer Science, Blekinge Institute of Technology, 2002.
34. Roger M. Needham and Michael D. Schröeder. Using encryption for authentication in large networks of computers. *Commununicatoins of the ACM*, 21(12):993–999, 1978. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359657.359659.
35. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
36. OMG. MOF Core v. 2.0, . Document Id: formal/06-01-01. `http://www.omg.org`.

37. OMG. MOF Models to Text Transformation Language Final Adopted Specification, . Document: ptc/06-11-01. `http://www.omg.org`.
38. OMG. OCL Version 2.0, . Document id: formal/06-05-01. `http://www.omg.org`.
39. OMG. UML Infrastructure, . Document: formal/05-07-05. `http://www.omg.org`.
40. OMG. UML: Superstructure. Version 2.0, . Document id: formal/05-07-04. `http://www.omg.org`.
41. Tahina Ramananandro. Mondex, an electronic purse: specification and refinement checks with the alloy model-finding method. *Form. Asp. Comput.*, 20(1):21–39, 2007. ISSN 0934-5043. doi: http://dx.doi.org/10.1007/s00165-007-0058-z.
42. Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universitaet Bremen, 2002. Logos Verlag, Berlin, BISS Monographs, No. 14.
43. James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. The Addison-Wesley Object Technology Series. Addison-Wesley, Harlow, England, 1999.
44. Ilya Shlyakhter, Robert Seater, Daniel Jackson, Manu Sridharan, and Mana Taghdiri. Debugging overconstrained declarative models using unsatisfiable cores. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering, Montreal, Canada*, pages 94–105. IEEE Computer Society, 2003.
45. Mana Taghdiri and Daniel Jackson. A lightweight formal analysis of a multicast key management scheme. In *Formal Techniques for Networked and Distributed Systems - FORTE 2003*, volume 2767 of *Lecture Notes in Computer Science*, pages 240–256. Springer, 2003.
46. TLSWG. SSL 3.0 specification, 1996. `http://wp.netscape.com/eng/ssl3`.
47. Emina Torlak, Marten van Dijk, Blaise Gassend, Daniel Jackson, and Srinivas Devadas. Knowledge flow analysis for security protocols. Technical Report MIT-CSAIL-TR-2005-066, Computer Science and Artificial Intelligence Laboratory, MIT, Oct. 2005. Available at:.
48. Mandana Vaziri and Daniel Jackson. Some Shortcomings of OCL, the Object Constraint Language of UML. In *Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*, pages 555–562, Santa Barbara, California, 2000.
49. Chris Wallace. Using Alloy in process modelling. *Information and Software Technology*, 45(15):1031–1043, December 2003. Publisher: Elsevier Science.
50. Manuel Wimmer and Gerhard Kramler. Bridging grammarware and modelware. In Jean-Michel Bruel, editor, *MoDELS Satellite Events*, volume 3844 of *Lecture Notes in Computer Science*, pages 159–168. Springer, 2006. ISBN 3-540-31780-5.
51. Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall, Upper Saddle River, NJ, USA, 1996.

A Alloy Metamodel Well-Formedness Constraints

A number of (mainly syntactic) well-formedness rules have been defined in accordance with the Alloy language specification [27]. The list of the well-formedness constraints specified here were inferred from the Alloy reference manual and may not be complete. Our implementation is using the Alloy Analyzer, which checks for such syntactic and semantic constraints. Therefore if a not well-formed Alloy model is generated, the Alloy Analyzer will flag an error, which will be displayed by our implementation. The purpose of the well-formedness constraints presented here, is to apply model transformation testing techniques in the future. The elements of the Alloy metamodel referenced here are depicted in Figure 3. More specifically:

– An element (i.e. signature, field) has to have a name:

```
context Id
inv: not self.name -> isEmpty()
    and self.name <> ‘‘’’
```

– All identifiers are unique:

```
context Id
inv: Id.allInstances -> forAll(i:Id |
    (i<>self) implies (i.name<>self.name)
```

– A signature cannot extend itself:

```
context ExtendsSigDecl
inv: ExtendsSigDecl.allInstances() -> forAll(
s:ExtendsSigDecl | (s <> self) implies
(s.extends.refers <> self))
```

Additionally signature extension is acyclic. We do not provide OCL for this constraint, because it requires recursion.

– A signature reference must reference a signature that has been declared:

```
context SigRef
inv: SigDecl.allInstances() -> exists(s:SigDecl |
    s.declares -> includes(self.refers))
```

B UML Profile for Alloy Constraints on the UML Metamodel

Our UML profile for Alloy extends the UML Kernel package metamodel elements with additional constraints. These constraints are presented here informally (i.e. using natural language). So as not to replicate the UML specification the elements of the Kernel diagram of the UML metamodel, which our profile uses, the Kernel diagrams are not present here; instead the reader is referred to the UML specification [40].

– A *NamedElement* is required to have a *name*.

- A *NamedElement* can not have a *name*, which is an Alloy keyword.
- The *visibility* attribute of a *NamedElement* is ignored during the transformation, because a similar concept does not exist in Alloy[3].
- Our transformation does not support package merge and package import. As a result a *NameSpace* can not be related to an *ElementImport* or a *PackageImport.* (see [40, p. 23])
- For a *MultiplicityElement*, *isOrdered* = *false* and *isUnique* = *true* always. Defining the transformation rules for *isOrdered* = *true* or *isUnique* = *false* remains for future work.
- The *ValueSpecification* of a *MultiplciityElement* can be either a *LiteralInteger* or a *LiteralUnlimitedNatural.* Thus the *lower* and *upper* multiplicity value of a *MultiplcitityElement* can not be any custom expression, as allowed by the UML standard.
- The UML metamodel allows for a *TypedElement* to be without a type [40, p. 24]. Our profile requires that every *TypedElement* has a type.
- The only *Constraint* expressions supported by our approach are *OpaqueExpressions* with OCL 2.0 as the *language.* All other Constraint expressions are ignored.
- A *Constraint* needs to have exactly one *Namespace.*
- Currently we only allow the definition of Constraints in the context of a Class. Constraints defined in other Contexts (i.e. *Package*) are not allowed.
- All constraints are required to have a unique name.
- *Class* names and *Property* names need to be unique.
- A *Package* name cannot be the same as the name of any other *NamedElement.*
- The *isLeaf* metaatribute is ignored.
- We do not allow the definition of *static* attributes, association ends or operations.
- A *Classifier* can be related to only one *general* Classifier. The *general* association end, relates the *Classifier* with its *direct* [40, p. 26] ancestors. This constraint forbids multiple inheritance.
- The *Generalization isSubstitutable* metaatribute is ignored.
- A *BehavioralFeature* may not specify any *raisedExceptions.*
- A *Parameter* of an operation can only be *inout* or *return.* No other kinds of parameters are allowed.
- An Operation may not be redefined. More formally:

```
context Operation
 inv: self.redefinedOperation -> isEmpty()
```

- Similarly an Attribute may not be *redefined.*
- A Class may not own *nestedClassifiers* (i.e. no inner classes are allowed).

[3] Recently the notion of visibility has been added to the Alloy language, so it might be possible to map UML visibility to Alloy. However this remains for future research.

- If a Class is *abstract* it is required that it is directly or indirectly extended by at least one concrete Class.
- A *Property* is *readOnly* by default. The *dynamic* stereotype can be used to override this constraint.
- The *Property aggregation* is *NONE*. No Property can be defined, which is aggregate or composite.
- The only *PrimitiveTypes* allowed in the model are Integers.
- A *Package* may not merge another Package.
- Currently our transformation does not deal with Package hierarchy. As a result, no *nestedPackages* are allowed.
- A *Package* may *own Classes*, *Associations*, *Generalizations*, or *DataTypes*, but no other metaelement.
- An *AssociationClass* cannot be specified in the class diagram.
- An *Association* may not be *Abstract*.
- An *Association* may not be redefined.
- Only binary Associations are allowed.
- *Qualified* association ends are not allowed.