# Translating Alloy Specifications to UML Class Diagrams Annotated with OCL

Ana Garis[1], Alcino Cunha[2], and Daniel Riesco[1]

[1] Universidad Nacional de San Luis, San Luis, Argentina
`{agaris,driesco}@unsl.edu.ar`
[2] DI-CCTC, Universidade do Minho, Braga, Portugal
`alcino@di.uminho.pt`

**Abstract.** Model-Driven Engineering (MDE) is a Software Engineering approach based on model transformations at different abstraction levels. It prescribes the development of software by successively transforming models from abstract (specifications) to more concrete ones (code). Alloy is an increasingly popular lightweight formal specification language that supports automatic verification. Unfortunately, its widespread industrial adoption is hampered by the lack of an ecosystem of MDE tools, namely code generators. This paper presents a model transformation between Alloy and UML Class Diagrams annotated with OCL. The proposed transformation enables current UML-based tools to also be applied to Alloy specifications, thus unleashing its potential for MDE.

**Keywords:** MDE, Alloy, UML, OCL

## 1 Introduction

Model-Driven Engineering (MDE) is a promising Software Engineering approach using models at different abstraction levels. Software is developed by successively transforming models from abstract to more concrete ones.

UML and OCL have been successfully adopted in the MDE context through the Model-Driven Architecture (MDA) initiative [15]. In order to support UML and OCL in MDE, different tools have been developed such as code generators and reverse engineering tools. Due to the informality and ambiguity of UML semantics it also has been combined with formal methods to increase the confidence in the software development process. Formal methods use mathematics for specification and design of models helping to discover inconsistencies in informal requirements. The main disadvantage of formal languages is that they require a learning effort and thus are frequently avoided by software engineers responding to time and cost constraints.

Alloy [12] is a lightweight formal language with a simple notation, easy to learn, easy to use, that includes a friendly Validation and Verification (V&V) tool. Its denotational language is based on first-order relational logic, with an object-oriented notation similar to UML and OCL [16]. The automatic Alloy

Analyzer allows the generation of snapshots showing instances of the model as well as the execution of operations and assertion checking.

Although very few UML software developers are familiar with formal methods, Alloy could be easily adopted by UML practitioners due to its simplicity and its resemblance with UML. Both Alloy and UML can benefit if two-way transformations are developed between them. On the one hand, from the UML practitioners point of view, Alloy Analyzer could be exploited as a model verification tool in a MDE context. On the other hand, from the Alloy practitioners point of view, a myriad of UML tools could be used in order to unleash Alloy potential for MDE. Specifically, there exist multiple code generators to different platforms and programming languages, such as JEE, CORBA, Java, C, C++, C# and Python, that could be used to refine Alloy specifications.

Further benefits could be achieved by developers familiarized with both Alloy and UML. They could be combined in the software development process: start by using UML Class Diagrams to specify requirements at high abstraction level, then translate them to Alloy and formally specify invariants and operations, perform model validation and verification using Alloy Analyzer, and finally translate back to UML+OCL in order to use the aforementioned code generation tools.

This paper presents a model transformation from Alloy specifications to UML Class Diagrams annotated with OCL. Although the semantic correspondence between elements of UML and Alloy was already analyzed in [1], the translation from Alloy to UML+OCL has not been considered yet. This translation opens new challenges since several Alloy expressions do not have a direct equivalent in UML or OCL. Additionally, it requires us to explore the different Alloy idioms in order to identify a specification style compatible with UML+OCL models. Therefore, we define a subset of the Alloy language which includes UML+OCL compatible expressions, and we study the semantics of the syntactic elements of Alloy, UML Class Diagrams and OCL. We redefine the EBNF of Alloy grammar to recognize expressions in this subset and specify the transformation rules. Our approach is illustrated with a case study.

The rest of the document is structured as follows. Next section introduces preliminary concepts related to Alloy. After discussing some related work, the transformation from Alloy to UML+OCL is presented. Last section presents the conclusions and future work.

## 2 Alloy

Alloy is a formal language based on first-order relational logic [12]. It is supported by a SAT solver that enables model V&V. Alloy Analyzer is inspired by model checkers, but it is implemented as a solver. An Alloy module consists of a module header, a set of imports and zero or more paragraphs. The *module header* is a name of the module where signatures, constraints, assertions and commands are defined. An *import* allows to include additional modules. Furthermore, a *paragraph* can either be a signature declaration, a constraint, an assertion or a command.

A *signature declaration* represents a set of atoms. An atom is a unity with three fundamental properties: it is indivisible, immutable and uninterpreted. Optionally, a signature declaration can introduce *fields*. Fields represent sets of tuples of atoms and are interpreted as relations between signatures. *Constraints* are defined by *facts*, *predicates* and *functions*. Facts are invariants; i.e, their associated constraints always hold. Predicates are named constraints, which can be used in diverse contexts. The difference between a fact and a predicate is that the first one always holds while the second one only holds when invoked. Finally, functions describe named expressions, which can be also reused in diverse contexts. *Assertions* allow the expression of properties that are expected to hold as consequence of the stated facts. *Commands* are instructions to perform particular analysis. Alloy provides two commands for analysis: `run` and `check`. Command `run` gives instructions to analyzer to search for an instance of a given predicate, and command `check` to search for a counterexample of a given assertion.

Alloy's logic is quite generic and does not commit to a particular specification style. For example, since atoms are immutable there is no standard way to model the dynamic behavior of operations, and several idioms have been proposed to address this issue. One of the most popular is to introduce a signature denoting the overall state of the system, and model operations as predicates that specify the relationship between pre- and post-states. Two variants of this idiom are possible, known respectively as *global state* and *local state*. In the former all mutable fields are defined in the global state signature. In the later, the state signature is added locally as an extra column at the end of each mutable field. The local state idiom is more modular, since fields are still declared in the signature they naturally belong to. On the other hand, the global state idiom forces all dynamic fields to be artificially grouped together. The designation *local state* can be misleading, since the state is also global - the "local" concerns only the location it appears in field declarations.

In this paper we will assume the local state idiom to specify operations. Note that Alloy models conforming to the global state idiom can be easily converted to the local one using a simple refactoring [9]. Without loss of generality, we will also assume the distinguished state signature to be denoted as `Time` and all fields to be dynamic. An operation `op` will be specified using a predicate `pred op[...,t,t':Time] {...}` with two special parameters `t` and `t'` denoting, respectively, the pre- and post-state. Functions will be used to model queries that do not change the state. As such, only one of those special parameters is needed.

Figure 1 presents an example of an Alloy model conforming to the local state idiom. It is a variant of the address book model first presented in [12]. The `addr` field is a mutable relation that maps names to targets. A target is either an address or another name. Names are either groups or aliases. For each book, the first fact forces all names in the `addr` relation to be registered in the respective `names` relation.

In Alloy *everything is a relation*. For example, variables are just unary singleton relations. As such, the relational composition operator can be used for

```
module addressBook

sig Time {}

abstract sig Target { }
sig Addr extends Target { }
abstract sig Name extends Target { }
sig Alias, Group extends Name { }

sig Book {
   names: Name set -> Time,
   addr: Name -> some Target -> Time }

fact { all t:Time | all b:Book | b.addr.t.Target in b.names.t }

fact { all t:Time | all b:Book | no n: Name | n in n.^(b.(addr.t)) }

fact { all t:Time | all b:Book | all a: Alias | lone a.(b.addr.t)  }

pred add [b: Book, n: Name, a: Target, t,t': Time] {
   n in b.names.t
   b.addr.t' in b.addr.t + n->a }

fun lookup [b: Book, n: Name, t: Time] : set Addr {
   n.^(b.(addr.t)) & Addr }
```

**Fig. 1.** Address book example.

various purposes. In particular, `b.addr.t` denotes the value of the relation `addr` of book `b` at instant `t`. Note that the relation `b.addr.t` has type `Name -> some Target`. If we compose it with the `Target` set, we get all names in the domain of that relation. The second fact uses the transitive closure operator to ensure that the `addr` relation is acyclic. The last fact limits the addresses of aliases to at most one target.

In the body of operations, constraints that do not refer to `t'` can be seen as pre-conditions. For example, `n in b.names.t` requires names to be registered before adding or removing new targets. Otherwise we have post-conditions. For example, expression `b.addr.t' = b.addr.t + n->a` states that, after executing operation `add`, relation `b.addr` should have at least one additional tuple. `lookup` models a query that returns the set of addresses of a given name. Again, transitive closure is used to recursively traverse relation `addr`. A desirable assertion in this model could be:

```
assert lookupYields {
   all t:Time, b:Book, n:b.names.t | some lookup[b,n,t]
}
check lookupYields for 4 but 1 Book
```

Unfortunately it does not hold, and the `check` command will produce a counter-example.

## 3 Alloy and UML+OCL Integration in MDE

The translation from Alloy to UML could foster the usage of Alloy in the MDE context. As mentioned before, a lot of UML tools have been developed to support MDA. However, even though OCL tools have improved in the last years, they still have several limitations regarding model V&V. Formal methods have been proposed as a valuable alternative to improve these limitations. In fact, they have been successfully integrated in the MDA, as shown in HOL-OCL [3], USE [11] or UML2Alloy [1].

Due to its suitability for V&V, Alloy is a better candidate for the early modeling phase of the software development process. After the validation process with Alloy, models can be translated to UML Class Diagrams and OCL in order to enable MDA-UML tools as well as MDA-OCL tools. Most of the UML tools allow transformations from UML Class Diagrams to different platforms and programming languages, such as JEE, CORBA, Java, C, C++, C# and Python. Additionally, there exist OCL tools for code generation, such as OCLtoSQL [4] and DresdenOCL [5].

### 3.1 Related Work

The relationship between UML and OCL with Alloy has been extensively studied by Anastaskis et al. [1], resulting in a prototype named UML2Alloy. This translation considers only the basic elements of UML Class Diagrams: classes, attributes and associations; and excludes interfaces, dependencies and signals.

Massoni et al. also propose a UML+OCL to Alloy translation in [13]. For classes, attributes and associations, they propose the same approach as Anastaskis et al. [1], but they also consider the translation of UML interfaces. The semantics of UML cannot be fully preserved since Alloy cannot represent some UML Class Diagram features such as visibility of attribute's properties.

UML has also been mapped to Alloy for model V&V of particular case-studies. We present three examples: the first one uses the Alloy Analyzer for formal security evaluation in a methodology called Aspect-Oriented Risk-Driven Development (AORDD) [7]; the second one describes a proposal for Alloy specification from Aspect-UML models, a UML Profile for extending UML with Aspect-oriented concepts [14]; the third one explains an approach to translate UML models, specified with OntoUML, for model validation using Alloy [2]. These examples make evident Alloy potential to V&V, but like in [13, 1]; they consider the translation from UML+OCL to Alloy, not from Alloy to UML.

Shah et al. present a model transformation from Alloy to UML [18]. Specifically, they convert instances generated by Alloy Analyzer to a UML Object Diagram. This approach is possible in case UML2Alloy tool has been used before to generate an Alloy specification. The transformation is based on the original

UML Class Diagram so it assumes that Alloy specifications must not change. Even though Shah et al. have exposed a way to translate from Alloy to UML, they only map model instances to UML Object Diagrams. Namely, they do not consider the translation from Alloy to OCL either.

## 4 Model Transformation from Alloy to UML+OCL

Although model transformations from UML to Alloy have already been defined [1, 13], the opposite translation has not. Moreover it is characterized by new challenges, not addressed previously. An important issue to solve is the characterization of source models, namely identifying a subset of Alloy that can faithfully be translated to UML+OCL. We will first define this subset formally, and then present the translation of both model declarations and constraints. The address book example presented in Figure 1 will be used to illustrate our proposal.

### 4.1 Characterizing source models

The Alloy subset accepted by our model transformation is defined in Figure 2. This subset restricts models to conform to the local state idiom, informally introduced in Section 2. In particular, the model must declare a distinguished signature denoted `Time`, the last column of field declarations must be `Time`, all predicates must have `t` and `t'` as parameters, and all functions must have `t` as parameter. Note that `Time`, `t`, and `t'` are reserved keywords not allowed in *sigId* and *varId*, respectively. To simplify the presentation, we are assuming all relations to be mutable. Immutable relations (not including `Time` in their declaration) could also be handled by adding frame conditions to all method post-conditions in OCL, stating that their value remains constant. Besides these structural restrictions, the syntax of formulas is further restricted to ensure a sound operational semantics [10]:

- Facts must be of the form `all t:Time |` $\phi$, with `t` the only time variable that occurs in formula $\phi$. This ensures that they act as invariants, instead of arbitrary temporal formulas, and can thus be represented in OCL.
- Every relational expression occurring in a formula must be *state-bound*, in the sense that each mutable relation identifier is within scope of a time variable. To simplify the translation, we ensure this restriction by forcing relation identifiers to be composed with either `t` or `t'`. However, a more relaxed syntax can be defined, where each occurrence of a relation identifier is required to be a subterm of either $\Phi$`.t` or $\Phi$`.t'`, where $\Phi$ denotes a relational expression.

Besides conforming to the local state idiom, an Alloy model must satisfy some additional restrictions due to the limitations of UML+OCL, as described in the UML Class Diagram metamodel [17] and the OCL metamodel [16]:

```
     module ::= module moduleId sig Time {} sigDecl+ paragraph*
  paragraph ::= factDecl | funDecl | predDecl
    sigDecl ::= [abstract] sig sigId [extends sigId] sigBody
    sigBody ::= { [fieldDecl (, fieldDecl)*] }
  fieldDecl ::= setDecl | relDecl
    setDecl ::= relId : set Time
    relDecl ::= relId : sigId [mult] -> Time | relId : (sigId ->)+[mult] sigId -> Time
       mult ::= lone | one | some | set
   factDecl ::= fact {all t:Time | all varId:sigId | form}
    funDecl ::= fun funId[(varId:sigId,)+t:Time] : set sigId { expr }
   predDecl ::= pred predId[(varId:sigId,)+t,t':Time] { form+ }
       form ::= expr compOp expr | form logicOp form | form => form [, form] |
                !form | intExpr intCompOp intExpr | quant varId:sigId | form
    logicOp ::= && | || | <=>
     compOp ::= = | in
  intCompOp ::= = | < | > | =< | >=
      quant ::= all | no | lone | one | some
       expr ::= varId | sigId | relId.t | relId.t' | none | univ | iden | expr relOp expr |
                ~expr | ^((varId.)*(relId.t)) | ^((varId.)*(relId.t')) |
                *((varId.)*(relId.t)) | *((varId.)*(relId.t')) | funId[(varId,)+t] |
                funId[(varId,)+t'] | {varId:sigId(,varId:sigId)+ | form}
      relOp ::= . | + | - | & | <: | :> | -> | ++
    intExpr ::= integer | #expr | int[expr] | intExpr intOp intExpr
      intOp ::= + | -
```

**Fig. 2.** Subset of Alloy translated to UML+OCL.

– Signature declarations can only be top-level or extend other signatures. Signature inclusion, where `in` is used instead of `extends`, is not allowed since it is not possible to specify using UML class diagrams that a class is a non-disjoint subset of another class.
– Field declarations must refer signature identifiers instead of arbitrary relational expressions. This ensures that the type of each column corresponds to a single signature, instead of an arbitrary disjunction of signatures, as prescribed in the Alloy type system [6]. Since fields will be represented by attributes or associations in UML, without this restriction we might not be able to determine the type of attributes or association ends.
– Multiplicity constraints can only occur in the last column (not counting `Time`) of a field declaration. A field relating more than two signatures will

be represented by a qualified association in UML, and those only support multiplicities in the association end.
- OCL requires a *context* (a class) for all invariants and methods. As such, Alloy facts must be further restricted to include at least one additional universally quantified variable besides the special time variable. The type of this variable will determine the OCL context. Moreover, functions and predicates are required to have at least one parameter besides the special time parameters. The type of the first parameter will determine the context of the target method.
- OCL does not natively support transitive closure. So far, we only managed to translate the transitive closure of a concrete relation, instead of an arbitrary relational expression. The syntax is restricted accordingly.
- Predicate call is not supported, since OCL constraints can only invoke side-effect free queries.
- Assertions and commands will not be considered, since they do not have a counterpart in OCL. In general they only make sense for model V&V, for which OCL is currently not well suited. We prescribe that model V&V should be performed using the Alloy Analyzer, so those constructs can safely be ignored when translating to OCL.

The grammar of Figure 2 also includes some additional minor restrictions, that do not limit the expressivity of the language. For example, we do not allow signature facts, neither atomic formulas of the form *mult expr*. These restrictions simplify the presentation of the translation, and can easily be lifted by means of trivial refactorings. For instance, the formula `lone a.(b.addr.t)` in Figure 1 can be refactored to `#(a.(b.addr.t))=<1`.

### 4.2 From Alloy to UML class diagrams

The relationship between UML class diagrams and Alloy declarations is straightforward, as noticed in [19, 13, 1]: in general, classes correspond to signatures (preserving the inheritance relation), associations and attributes to fields, and methods to predicates and functions. These relationships are essentially the same when translating from Alloy to UML class diagrams, but with the novelty that some fields may lead to non-binary associations.

As seen in Figure 2, excluding the mandatory `Time` column, fields can be of two kinds:

- Sets with type $A$, to be translated as attributes of class $A$ with type `Boolean`.
- Relations with type $A_1$ `->` ... `->` $m$ $A_n$, to be translated as qualified associations between $A_1$ and $A_n$, with $A_2, \ldots, A_{n-1}$ as qualifiers. The multiplicity at the end of the association depends on $m$: `0..*` for `set`; `1..*` for `some`; `0..1` for `lone`; and `1..1` for `one`. If $m$ is absent, the default is `set`.

If the relation is binary, with type $A_1$ `->` $m$ $A_2$, and $m$ is either `lone` or `one` it is more natural to encode it as attribute of $A_1$ with type $A_2$.

The UML class diagram corresponding to the address book example of Figure 1 is presented in Figure 3.

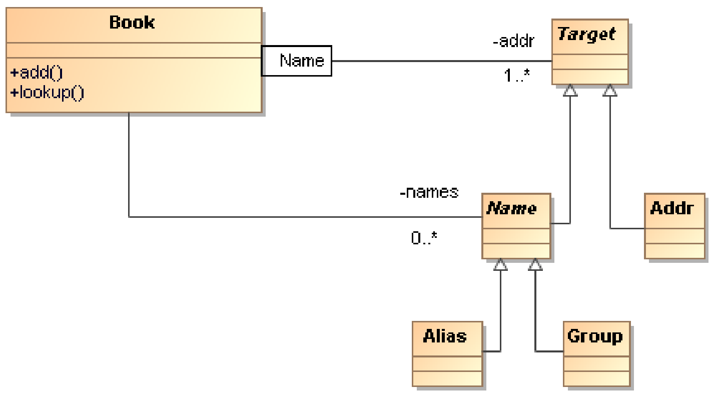

**Fig. 3.** UML class diagram corresponding to address book example

## 4.3 From Alloy to OCL

The model transformation from the Alloy subset described in Section 4.1 to OCL will be encoded using an embedding function $[\![\cdot]\!]$. To simplify the presentation, this function will accept and produce concrete syntax. The following convention will be followed for naming variables denoting the various grammar elements: $x, y, z$ for variable identifiers; $A, B, C$ for signature identifiers and types in general; $R, S, T$ for relation identifiers; $\phi, \psi, \varphi$ for formulas; $\Phi, \Psi, \Upsilon$ for relational expressions; and $\alpha, \beta, \gamma$ for integer expressions.

We will assume the input model to be well-typed, according to the typing system described in [6]. This type system is very relaxed: an error occurs when a expression can be shown to always be empty at static time. For example, the composition $\Phi$`.`$\Psi$ is well-defined for any relational expressions $\Phi$`:` $A$ `->` $B$ and $\Psi$`:` $C$ `->` $D$, if the intersection of types $B$ and $C$ is non-empty. The type of a relational expression is itself a relation: a set of tuples of atomic signatures (i.e. signatures that are not further extended, such as `Book`, `Addr`, `Alias`, and `Group` in our running example). The type inference rules ensure that all the tuples in the type relation have the same length. Given a relational expression $\Phi$ of arity $|\Phi|$, we will denote the type of the $n$-th column as $\Phi^n$ (assuming $0 < n \leqslant |\Phi|$). The type of a column is guaranteed to be a set of atomic signatures, each corresponding to a concrete class in UML. In the translation we sometimes need to quantify over such types. To simplify the presentation, notation

$$\{A_1, \ldots, A_n\}\texttt{.allInstances->forAll(}x\,\texttt{|}\,\phi\texttt{)}$$
$$\{A_1, \ldots, A_n\}\texttt{.allInstances->exists(}x\,\texttt{|}\,\phi\texttt{)}$$

where $\{A_1, \ldots, A_n\}$ is an Alloy type, will be used as a shorthand for, respectively

$$A_1\texttt{.allInstances->forAll(}x\,\texttt{|}\,\phi\texttt{) and } \ldots \texttt{ and } A_n\texttt{.allInstances->forAll(}x\,\texttt{|}\,\phi\texttt{)}$$
$$A_1\texttt{.allInstances->exists(}x\,\texttt{|}\,\phi\texttt{) or } \ldots \texttt{ or } A_n\texttt{.allInstances->exists(}x\,\texttt{|}\,\phi\texttt{)}$$

The translation of an Alloy module is triggered by the following rule:

$$[\![\texttt{module}\ id\ \texttt{sig Time}\ \{\}\ s_1 \ldots s_n\ p_1 \ldots p_m]\!] \equiv$$
$$\texttt{package}\ m\ [\![p_1]\!] \ldots [\![p_m]\!]\ \texttt{endpackage}$$

**Fact, Function and Predicate Declarations** Figure 4 details the transformations of fact, function and predicate declarations. Signature declarations are ignored in the OCL generation, and are only used in the UML class diagram generation detailed in the previous section.

$[\![\texttt{fact}\ \{\texttt{all t:Time | all}\ x\texttt{:}A\ \texttt{|}\ \phi\}]\!] \equiv$
    $\texttt{context}\ A\ \texttt{inv:}\ [\![\phi]\!]_x$
$[\![\texttt{fun}\ f\texttt{[}x_1\texttt{:}A_1\texttt{,}\ldots\texttt{,}x_n\texttt{:}A_n\texttt{,t:Time]}\ \texttt{:}\ \texttt{set}\ B\ \{\ \Phi\ \}]\!] \equiv$
    $\texttt{context}\ A_1\texttt{::}f\texttt{(}x_2\texttt{:}A_2\texttt{,}\ldots\texttt{,}x_n\texttt{:}A_n\texttt{):Set(}B\texttt{)}$
        $\texttt{post:}\ B\texttt{.allInstances->select(}y\,|\,[\![\langle y\rangle \in \Phi]\!]_{x_1}\texttt{)}$
$[\![\texttt{pred}\ f\texttt{[}x_1\texttt{:}A_1\texttt{,}\ldots\texttt{,}x_n\texttt{:}A_n\texttt{,t,t':Time]}\ \{\ \phi_1 \ldots \phi_m\ \}]\!] \equiv$
    $\texttt{context}\ A_1\texttt{::}f\texttt{(}x_2\texttt{:}A_2\texttt{,}\ldots\texttt{,}x_n\texttt{:}A_n\texttt{)}$
        $\texttt{pre:}\ [\![\phi_1]\!]_{x_1}$ if `t'` does not occur in $\phi_1$
        $\texttt{post:}\ [\![\phi_1]\!]'_{x_1}$ otherwise
        ...
        $\texttt{pre:}\ [\![\phi_m]\!]_{x_1}$ if `t'` does not occur in $\phi_m$
        $\texttt{post:}\ [\![\phi_m]\!]'_{x_1}$ otherwise

**Fig. 4.** Translation of fact, function and predicate declarations

In OCL, all invariants and method specifications must be defined in the context of a class. For Alloy facts, the type of the first universally quantified variable (appart from the mandatory `Time` one) will determine the context of the generated invariant. The translation of formulas must then be parametrized with the name of the variable that will denote the `self` object. For functions and predicates, the context is determined by the type of the first parameter. In a predicate, all formulas where `t'` does not occur will be translated as pre-conditions. Otherwise, they are translated as post-conditions.

Two slightly different formula translations will be defined, due to different meanings that variable `t` assumes in different contexts. In a post-condition, an association $R$`.t` should be translated as $R$`@pre`, since `t` denotes the pre-state. In invariants, functions and pre-states it denotes the only visible state, and thus the translation just outputs $R$. As such, we will use $[\![\phi]\!]$ to translate a formula $\phi$ that occurs in an invariant, function, or pre-condition; and $[\![\phi]\!]'$ to translate a formula $\phi$ that occurs in a post-condition.

**Formulas** The translation of formulas is presented in Figure 5. We omit the definition of $[\![\cdot]\!]'$ because for formulas it is identical - it will only diverge when applied to relational expressions. Most logic operators have a direct counterpart in OCL and can thus be trivially translated. OCL does not support the non-standard quantifiers `no` and `one`, but they can be simulated by testing the cardinality of the subset of the type satisfying the quantified formula.

$$
\begin{aligned}
[\![\Phi \texttt{ in } \Psi]\!]_x &\equiv \Phi^1\texttt{.allInstances->forAll(}y_1\texttt{|}\ldots\\
&\quad\ \Phi^{|\Phi|}\texttt{.allInstances->forAll(}y_{|\Phi|}\texttt{|}\\
&\qquad [\![\langle y_1,\ldots,y_{|\Phi|}\rangle \in \Phi]\!]_x \texttt{ implies } [\![\langle y_1,\ldots,y_{|\Phi|}\rangle \in \Psi]\!]_x\texttt{)}\ldots\texttt{)}\\
[\![\Phi \texttt{ = } \Psi]\!]_x &\equiv [\![\Phi \texttt{ in } \Psi]\!]_x \texttt{ and } [\![\Psi \texttt{ in } \Phi]\!]_x\\
[\![\phi \texttt{ \&\& } \psi]\!]_x &\equiv [\![\phi]\!]_x \texttt{ and } [\![\psi]\!]_x\\
[\![\phi \texttt{ || } \psi]\!]_x &\equiv [\![\phi]\!]_x \texttt{ or } [\![\psi]\!]_x\\
[\![\phi \texttt{ <=> } \psi]\!]_x &\equiv [\![\phi \texttt{ => } \psi]\!]_x \texttt{ and } [\![\psi \texttt{ => } \phi]\!]_x\\
[\![\phi \texttt{ => } \psi]\!]_x &\equiv [\![\phi]\!]_x \texttt{ implies } [\![\psi]\!]_x\\
[\![\phi \texttt{ => } \psi\texttt{,}\varphi]\!]_x &\equiv \texttt{if } [\![\phi]\!]_x \texttt{ then } [\![\psi]\!]_x \texttt{ else } [\![\varphi]\!]_x \texttt{ endif}\\
[\![\texttt{!}\phi]\!]_x &\equiv \texttt{not } [\![\phi]\!]_x\\
[\![\alpha \texttt{ = } \beta]\!]_x &\equiv [\![\alpha]\!]_x \texttt{ = } [\![\beta]\!]_x\\
&\ \ \ldots\\
[\![\alpha \texttt{ >= } \beta]\!]_x &\equiv [\![\alpha]\!]_x \texttt{ >= } [\![\beta]\!]_x\\
[\![\texttt{all } y\texttt{:}A \texttt{ | } \phi]\!]_x &\equiv A\texttt{.allInstances->forAll(}y\texttt{|}[\![\phi]\!]_x\texttt{)}\\
[\![\texttt{some } y\texttt{:}A \texttt{ | } \phi]\!]_x &\equiv A\texttt{.allInstances->exists(}y\texttt{|}[\![\phi]\!]_x\texttt{)}\\
[\![\texttt{no } y\texttt{:}A \texttt{ | } \phi]\!]_x &\equiv A\texttt{.allInstances->select(}y\texttt{|}[\![\phi]\!]_x\texttt{)->isEmpty()}\\
[\![\texttt{one } y\texttt{:}A \texttt{ | } \phi]\!]_x &\equiv A\texttt{.allInstances->select(}y\texttt{|}[\![\phi]\!]_x\texttt{)->size() = 1}
\end{aligned}
$$

**Fig. 5.** Translation of formulas

The trickiest part of the translation concerns the atomic formulas $\Phi$ `in` $\Psi$, where $\Phi$ and $\Psi$ are arbitrary relational expressions. This formula cannot be encoded using set inclusion because $|\Phi|$ can be greater than 1, and, unlike Alloy, OCL does not support the construction of arbitrary relations as normal first-order values. As such, relational expressions will be translated by building their standard first-order denotational semantics: a relational expression $\Phi$ will be translated by a formula that checks if a tuple $\langle y_1,\ldots,y_{|\Phi|}\rangle$ belongs to the denoted relation. The inclusion $\Phi$ `in` $\Psi$ can thus be translated by a formula that checks if all tuples of the appropriate type that belong to $\Phi$ also belong to $\Psi$. Note that the type system ensures that the arity of $\Phi$ and $\Psi$ are the same. Using the first-order semantics of relational logic to embed Alloy in other logical frameworks is kind of folklore. For example, a similar strategy was used recently to develop a tool for unbounded verification of Alloy models using SMT solvers [8].

**Relational Expressions** The translation of relational expressions is presented in Figure 6. As explained above, this translation basically encodes the standard first-order semantics of relational operators. A brief explanation of the most interesting rules follows:

- Testing if a unary tuple is a member of a variable can be done with a simple equality test. Note that, as mentioned before, Alloy variables are singleton unary relations. If the variable denotes the `self` object then this identifier is used instead.
- Translation of field $R$`.t` membership depends on the arity of $R$: if it is a set we just check the value of the generated boolean attribute; otherwise we navigate the qualified association $R$.
- Translation of field membership requires an additional type checking since Alloy allows access to a field from any signature that includes the owner of the field. A reference like this could generate an undefined value in OCL. As such, we translate to `false` when the type of each variable $y_i$ is not a sub-type of $R^i$. Translation of variables and signature membership assumes similar considerations. To simplify the presentation, these type-checkings are not included in Figure 6.
- The semantics of the relational composition $\Phi$`.`$\Psi$ leads to a new existential quantifier over the mediating type. We quantify over $\Phi^{|\Phi|} \cap \Psi^1$ because the composition only succeeds for values belonging to the intersection of both types. This optimization reduces the number of quantifiers in the outputted formula.
- Testing if $\langle y_1, \ldots, y_n \rangle$ is included in the relational overriding $\Phi$ `++` $\Psi$ is reduced to a membership test over $\Psi$ if $\langle y_1, \ldots, y_{n-1} \rangle$ belongs to its domain; otherwise a membership test over $\Phi$ is generated.
- In a relation defined by the comprehension $\{z_1$`:`$A_1$`,...,`$z_n$`:`$A_n$ `|` $\phi\}$, the membership test is translated by just applying the predicate $\phi$ to the tuple variables $y_1, \ldots, y_n$ instead of $z_1, \ldots, z_n$.

The translation of closures is not straightforward since they are not finitely axiomatizable in first order logic, and OCL also does not support them natively. Fortunately, there are many ways to encode the transitive closure using recursive definitions. Given an arbitrary relation $R$`:` $A_1$ `-> ... ->` $A_n$, the transitive closure of the respective (partially applied) qualified association can be implemented as follows:

```
context A₁
  def: RClosureAux(y₂:A₂,...,yₙ₋₁:Aₙ₋₁,s:Set(Aₙ)):Set(Aₙ) =
    let s':Set(Aₙ) = s->collect(x:A₁ | yₙ₋₁.R[yₙ₋₂,...,y₂,x] in
      if s->includesAll(s') then s
      else RClosureAux(y₂,...,yₙ₋₁,s->union(s'))
      endif
  def: RClosure(y₂:A₂,...,yₙ₋₁:Aₙ₋₁):Set(Aₙ) =
    RClosureAux(y₂,...,yₙ₋₁,yₙ₋₁.R[yₙ₋₂,...,y₂,self]))
```

$$\begin{aligned}
\llbracket \langle y\rangle \in z \rrbracket_x &\equiv \begin{array}{ll} y\texttt{=self} & \text{if } z = x \\ y\texttt{=}z & \text{otherwise}\end{array}\\
\llbracket \langle y\rangle \in A \rrbracket_x &\equiv A\texttt{.allInstances->includes(}y\texttt{)}\\
\llbracket \langle y_1,\ldots,y_n\rangle \in R\texttt{.t} \rrbracket_x &\equiv \begin{array}{ll} y_1\texttt{.}R\texttt{[}y_2\texttt{,}\ldots\texttt{,}y_{n-1}\texttt{]->includes(}y_n\texttt{)} & \text{if } n>1\\ y_1\texttt{.}R\texttt{()} & \text{otherwise}\end{array}\\
\llbracket \langle y\rangle \in \texttt{none} \rrbracket_x &\equiv \texttt{false}\\
\llbracket \langle y\rangle \in \texttt{univ} \rrbracket_x &\equiv \texttt{true}\\
\llbracket \langle y_1,y_2\rangle \in \texttt{iden} \rrbracket_x &\equiv y_1\texttt{=}y_2\\
\llbracket \langle y_1,\ldots,y_n\rangle \in \varPhi\texttt{.}\varPsi \rrbracket_x &\equiv (\varPhi^{|\varPhi|} \cap \varPsi^1)\texttt{.allInstances->exists(}y\texttt{|}\\
&\qquad \llbracket\langle y_1,\ldots,y_{|\varPhi|-1},y\rangle\in\varPhi\rrbracket_x \texttt{ and } \llbracket\langle y,y_{|\varPhi|},\ldots,y_n\rangle\in\varPsi\rrbracket_x\texttt{)}\\
\llbracket \langle y_1,\ldots,y_n\rangle \in \varPhi \texttt{ + } \varPsi \rrbracket_x &\equiv \llbracket\langle y_1,\ldots,y_n\rangle\in\varPhi\rrbracket_x \texttt{ or } \llbracket\langle y_1,\ldots,y_n\rangle\in\varPsi\rrbracket_x\\
\llbracket \langle y_1,\ldots,y_n\rangle \in \varPhi \texttt{ - } \varPsi \rrbracket_x &\equiv \llbracket\langle y_1,\ldots,y_n\rangle\in\varPhi\rrbracket_x \texttt{ and (not } \llbracket\langle y_1,\ldots,y_n\rangle\in\varPsi\rrbracket_x\texttt{)}\\
\llbracket \langle y_1,\ldots,y_n\rangle \in \varPhi \texttt{ \& } \varPsi \rrbracket_x &\equiv \llbracket\langle y_1,\ldots,y_n\rangle\in\varPhi\rrbracket_x \texttt{ and } \llbracket\langle y_1,\ldots,y_n\rangle\in\varPsi\rrbracket_x\\
\llbracket \langle y_1,\ldots,y_n\rangle \in \varPhi \texttt{ <: } \varPsi \rrbracket_x &\equiv \llbracket\langle y_1\rangle\in\varPhi\rrbracket_x \texttt{ and } \llbracket\langle y_1,\ldots,y_n\rangle\in\varPsi\rrbracket_x\\
\llbracket \langle y_1,\ldots,y_n\rangle \in \varPhi \texttt{ :> } \varPsi \rrbracket_x &\equiv \llbracket\langle y_1,\ldots,y_n\rangle\in\varPhi\rrbracket_x \texttt{ and } \llbracket\langle y_n\rangle\in\varPsi\rrbracket_x\\
\llbracket \langle y_1,\ldots,y_n\rangle \in \varPhi \texttt{ -> } \varPsi \rrbracket_x &\equiv \llbracket\langle y_1,\ldots,y_{|\varPhi|}\rangle\in\varPhi\rrbracket_x \texttt{ and } \llbracket\langle y_{|\varPhi|+1},\ldots,y_n\rangle\in\varPsi\rrbracket_x\\
\llbracket \langle y_1,\ldots,y_n\rangle \in \varPhi \texttt{ ++ } \varPsi \rrbracket_x &\equiv \texttt{if } \llbracket\langle y_1,\ldots,y_{n-1}\rangle\in\varPsi\texttt{.}\varPsi^n\rrbracket_x \texttt{ then } \llbracket\langle y_1,\ldots,y_n\rangle\in\varPsi\rrbracket_x\\
&\qquad \texttt{else } \llbracket\langle y_1,\ldots,y_n\rangle\in\varPhi\rrbracket_x \texttt{ endif}\\
\llbracket \langle y_1,\ldots,y_n\rangle \in \texttt{~}\varPhi \rrbracket_x &\equiv \llbracket\langle y_n,\ldots,y_1\rangle\in\varPhi\rrbracket_x\\
\llbracket \langle y_1,y_n\rangle \in \texttt{\^{}(}y_2\texttt{.}\ldots\texttt{.}y_{n-1}\texttt{.}R\texttt{.t)} \rrbracket_x &\equiv y_1\texttt{.}R\texttt{Closure(}y_2\texttt{,}\ldots\texttt{,}y_{n-1}\texttt{)->includes(}y_n\texttt{)}\\
\llbracket \langle y_1,y_n\rangle \in \texttt{*(}y_2\texttt{.}\ldots\texttt{.}y_{n-1}\texttt{.}R\texttt{.t)} \rrbracket_x &\equiv y_1\texttt{=}y_n \texttt{ or } \llbracket\langle y_1,y_n\rangle\in\texttt{\^{}(}y_2\texttt{.}\ldots\texttt{.}y_{n-1}\texttt{.}R\texttt{.t)}\rrbracket_x\\
\llbracket \langle y\rangle \in f\texttt{[}y_1\texttt{,}\ldots\texttt{,}y_n\texttt{,t]} \rrbracket_x &\equiv y_1\texttt{.}f\texttt{(}y_2\texttt{,}\ldots\texttt{,}y_n\texttt{)->includes(}y\texttt{)}\\
\llbracket \langle y_1,\ldots,y_n\rangle \in \{z_1\texttt{:}A_1\texttt{,}\ldots\texttt{,}z_n\texttt{:}A_n \mid \phi\} \rrbracket_x &\equiv \llbracket\phi[y_1/z_1,\ldots,y_n/z_n]\rrbracket_x
\end{aligned}$$

**Fig. 6.** Translation of relational expressions

The translation of relational expressions occurring in post-conditions is almost identical, with the exception of the rules presented in Figure 7, where relation identifiers within scope `t` are evaluated in the pre-state. $R$`ClosurePre` is an auxiliar definition similar to the one presented above, but with all occurrences of $R$ replaced by $R$`@pre`.

The blind application of these translation rules usually results in obfuscated OCL specifications, mainly due to the introduction of quantifiers in the translation of the relational inclusion and composition. Fortunately, some first-order equivalences can be applied to the resulting order to simplify it, namely:

$$\begin{aligned}
&A\texttt{.allInstances->exists(}y \mid y\texttt{=}z \texttt{ and } \phi\texttt{)} \equiv \phi[z/y]\\
&A\texttt{.allInstances->forAll(}y \mid y\texttt{=}z \texttt{ implies } \phi\texttt{)} \equiv \phi[z/y]
\end{aligned}$$

$$\llbracket\langle y_1,\ldots,y_n\rangle \in R\texttt{.t}\rrbracket'_x \equiv \begin{cases} y_1\texttt{.}R\texttt{@pre[}y_2\texttt{,}\ldots\texttt{,}y_{n-1}\texttt{]->includes(}y_n\texttt{)} & \text{if } n>1 \\ y_1\texttt{.}R\texttt{@pre()} & \text{otherwise}\end{cases}$$

$$\llbracket\langle y_1,\ldots,y_n\rangle \in R\texttt{.t'}\rrbracket'_x \equiv \begin{cases} y_1\texttt{.}R\texttt{[}y_2\texttt{,}\ldots\texttt{,}y_{n-1}\texttt{]->includes(}y_n\texttt{)} & \text{if } n>1 \\ y_1\texttt{.}R\texttt{()} & \text{otherwise}\end{cases}$$

$$\llbracket\langle y_1,y_n\rangle \in \texttt{\^{}(}y_2\texttt{.}\ldots\texttt{.}y_{n-1}\texttt{.}R\texttt{.t)}\rrbracket'_x \equiv y_1\texttt{.}R\texttt{ClosurePre(}y_2\texttt{,}\ldots\texttt{,}y_{n-1}\texttt{)->includes(}y_n\texttt{)}$$

$$\llbracket\langle y_1,y_n\rangle \in \texttt{\^{}(}y_2\texttt{.}\ldots\texttt{.}y_{n-1}\texttt{.}R\texttt{.t')}\rrbracket'_x \equiv y_1\texttt{.}R\texttt{Closure(}y_2\texttt{,}\ldots\texttt{,}y_{n-1}\texttt{)->includes(}y_n\texttt{)}$$

$$\llbracket\langle y_1,y_n\rangle \in \texttt{*(}y_2\texttt{.}\ldots\texttt{.}y_{n-1}\texttt{.}R\texttt{.t)}\rrbracket'_x \equiv y_1\texttt{=}y_n \texttt{ or } \llbracket\langle y_1,y_n\rangle \in \texttt{\^{}(}y_2\texttt{.}\ldots\texttt{.}y_{n-1}\texttt{.}R\texttt{.t)}\rrbracket'_x$$

$$\llbracket\langle y_1,y_n\rangle \in \texttt{*(}y_2\texttt{.}\ldots\texttt{.}y_{n-1}\texttt{.}R\texttt{.t')}\rrbracket'_x \equiv y_1\texttt{=}y_n \texttt{ or } \llbracket\langle y_1,y_n\rangle \in \texttt{\^{}(}y_2\texttt{.}\ldots\texttt{.}y_{n-1}\texttt{.}R\texttt{.t')}\rrbracket'_x$$

$$\llbracket\langle y\rangle \in f\texttt{[}y_1\texttt{,}\ldots\texttt{,}y_n\texttt{,t']}\rrbracket'_x \equiv y_1\texttt{.}f\texttt{(}y_2\texttt{,}\ldots\texttt{,}y_n\texttt{)->includes(}y\texttt{)}$$

**Fig. 7.** Translation of relational expressions in post-conditions

**Integer expressions** Figure 8 presents the translation of Alloy integer expressions to OCL. Alloy expression $\texttt{\#}\Phi$ computes the size of a relational expression $\Phi$ of arbitrary arity. The presented rule only works for unary expressions. Using tuples it is trivial to generalize it to arbitrary relational expressions.

$$\llbracket n\rrbracket_x \equiv n$$

$$\llbracket\texttt{\#}\Phi\rrbracket_x \equiv \Phi^1\texttt{.allInstances->select(}y\,|\,\llbracket\langle y\rangle \in \Phi\rrbracket_x\texttt{)->size()}$$

$$\llbracket\texttt{int[}\Phi\texttt{]}\rrbracket_x \equiv \Phi^1\texttt{.allInstances->select(}y\,|\,\llbracket\langle y\rangle \in \Phi\rrbracket_x\texttt{)->sum()}$$

$$\llbracket\alpha \texttt{ + } \beta\rrbracket_x \equiv \llbracket\alpha\rrbracket_x \texttt{ + } \llbracket\beta\rrbracket_x$$

$$\llbracket\alpha \texttt{ - } \beta\rrbracket_x \equiv \llbracket\alpha\rrbracket_x \texttt{ - } \llbracket\beta\rrbracket_x$$

**Fig. 8.** Translation of integer expressions

**The Address Book Case Study** An excerpt of the OCL model obtained from the address book example is presented in Figure 9. It includes the first two invariants and the specification of operation add. Both simplification rules where applied to the result, which was then manually rendered for better comprehension.

## 5 Concluding Remarks and Future Work

We have presented a model transformation from Alloy to UML class diagrams annotated with OCL. We have formally characterized the Alloy local state idiom accepted by the transformation. This idiom is sufficiently broad to encompass most specifications. When compared to the previously developed transformations

```
context Book inv:
  Name.allInstances->forAll(v0 |
    Target.allInstances->exists(v1 | self.addr[v0]->includes(v1)
         and Target.allInstances ->includes(v1))  implies
    self.names->includes(v0))

context Book inv:
  Name.allInstances->select(n |
    n.addrClosure(self)->includes(n))->isEmpty()

context Book::add(n:Name,a:Target)
  pre: self.names->includes(n)
  post: Name.allInstances->forAll(v0 | Target.allInstances->forAll(v1 |
          self.addr[v0]->includes(v1)  implies
          self.addr@pre[v0]->includes(v1) or (v0=n and v1=a)))
```

**Fig. 9.** OCL specifications of the address book example

from UML+OCL to Alloy [1, 13], this model transformation raised interesting new challenges, namely: the translation of relational expressions of arbitrary arity; dealing with the idiosyncrasies of Alloy's type-system; and the encoding of closures.

The transformation still has some limitations, most notably not allowing signature inclusion in the source Alloy model. Signature inclusion is mostly used in Alloy to overcome the single-inheritance limitation. We intend to extend our idiom to include such usages, and then translate it directly using multiple-inheritance. We also intend to extend it to other Alloy idioms that allow the specification of state-transition systems, in order to also generate UML state-chart diagrams.

We have implemented the proposed transformation in Haskell, generating syntax compatible with popular UML+OCL modeling applications. The tool is available for download at `http://sourceforge.net/projects/alloy2ocl`. It includes a couple of additional case-studies, but we intend to extend this set to further validate the transformation.

**Acknowledgments.** This research was partially supported by QREN (the Portuguese National Strategy Reference Chart) project 1621 – Evolve.

## References

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Software and Systems Modeling 9(1), 69–86 (2008)
2. Braga, B.F.B., Almeida, J.P.A., Guizzardi, G., Benevides, A.B.: Transforming OntoUML into Alloy: towards conceptual model validation using a lightweight formal method. Innovations in Systems and Software Engineering 6(1-2), 55–63 (2010)

3. Brucker, A.D., Wolff, B.: HOL-OCL: a formal proof environment for UML/OCL. In: Proceedings of Fundamental Approaches to Software Engineering. LNCS, vol. 4961, pp. 97–100. Springer-Verlag (2008)
4. Demuth, B., Hussmann, H., Loecher, S.: OCL as a specification language for business rules in database applications. In: UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools, LNCS, vol. 2185, pp. 104–117. Springer-Verlag (2001)
5. DresdenOCL website, `http://www.dresden-ocl.org/index.php/DresdenOCL`
6. Edwards, J., Jackson, D., Torlak, E.: A type system for object models. In: Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 189–199. ACM (2004)
7. Georg, G., Anastasakis, K., Bordbar, B., Houmb, S.H., Toahchoodee, I.R.M.: Verification and trade-off analysis of security properties in UML system models. IEEE Transactions on Software Engineering 36(3), 338–356 (2010)
8. Ghazi, A.A.E., Taghdiri, M.: Relational reasoning via SMT solving. In: Proceedings of the 17th International Symposium on Formal Methods. LNCS, vol. 6664, pp. 133–148. Springer-Verlag (2011)
9. Gheyi, R., Massoni, T., Borba, P.: Formally introducing Alloy idioms. In: Proceedings of the Brazilian Symposium on Formal Methods. pp. 22–37 (2007)
10. Giannakopoulos, T., J.Dougherty, D., Fisler, K., Krishnamurthi, S.: Towards an operational semantics for Alloy. In: Proceedings of the 16th International Symposium on Formal Methods. LNCS, vol. 5850, pp. 483–498. Springer-Verlag (2009)
11. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL models in USE by automatic snapshot generation. Software and Systems Modeling 4(4), 386–398 (2005)
12. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)
13. Massoni, T., Gheyi, R., Borba, P.: Formal refactoring for UML Class Diagrams. In: Proceedings of the 19th Brazilian Symposium on Software Engineering. pp. 152–167 (2005)
14. Mostefaoui, F., Vachon, J.: Verification of Aspect-UML models using Alloy. In: Proceedings of the 10th International Workshop on Aspect-Oriented Modeling. pp. 41–48. ACM (2007)
15. OMG: MDA Guide version 1.0.1 (2003)
16. OMG: Object Constraint Language, Version 2.2 (2010)
17. OMG: UML Superstructure, Version 2.3 (2010)
18. Shah, S.M.A., Anastasakis, K., Bordbar, B.: From UML to Alloy and back again. In: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation. ACM (2009)
19. Vaziri, M., Jackson, D.: Some shortcomings of OCL, the Object Constraint Language of UML. In: Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems. pp. 555–562. IEEE (2000)