Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Translating between Alloy Specifications and UML Class Diagrams Annotated with OCL

Alcino Cunha · Ana Garis · Daniel Riesco

Received: date / Accepted: date

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 from Alloy to UML Class Diagrams annotated with OCL (UML+OCL), and shows how an existing transformation from UML+OCL to Alloy can be improved to handle dynamic issues. The proposed bidirectional transformation enables a smooth integration of Alloy in current MDE contexts, by allowing UML+OCL specifications to be transformed to Alloy for validation and verification, to correct and possibly refine them inside Alloy, and to translate them back to UML+OCL for sharing with stakeholders or to reuse current Model-Driven Architecture (MDA) tools to refine them towards code.

Keywords MDE · Alloy · UML · OCL

Alcino Cunha
HASLab, INESC TEC and Universidade do Minho
Braga, Portugal
E-mail: alcino@di.uminho.pt

Ana Garis
Universidad Nacional de San Luis
San Luis, Argentina
E-mail: agaris@unsl.edu.ar

Daniel Riesco
Universidad Nacional de San Luis
San Luis, Argentina
E-mail: driesco@unsl.edu.ar

1 Introduction

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+OCL [30,31] have been successfully adopted in the MDE context through the MDA initiative [29]. In order to support UML+OCL in MDE, different tools have been developed such as code generators and reverse engineering tools. Due to the informality and ambiguity of the UML semantics it also has been combined with formal methods to increase the confidence in the software development process. Formal methods use semantically precise mathematical concepts to model software, helping the user to, for example, discover design flaws, inconsistencies in informal requirements, or the implications of such requirements. The main disadvantage of some formal methods, such as those involving interactive theorem proving, is that they require a considerable initial learning effort and thus are frequently avoided by software engineers responding to time and cost constraints.

In contrast with such "heavyweight" formal methods, Alloy [22] 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 specification language is based on first-order relational logic, with a familiar object-oriented notation. The automatic Alloy Analyzer allows the generation of snapshots showing instances of the model as well as assertion checking. Given the undecidability of Alloy's logic, such automatic analysis must be bounded by a user specified scope. However, it is still highly effective since most counterexamples can be found within small scopes.

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, the Alloy Analyzer could be exploited as a model verification tool in an 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's 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 familiar with both Alloy and UML. They could be combined in the software development process: start by using UML Class Diagrams (CDs) 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.

The main contribution of this paper is a model transformation from Alloy specifications to UML+OCL. Although the semantic correspondence between elements of UML and Alloy was already analyzed in [2, 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 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, CDs and OCL. We redefine the EBNF of Alloy's grammar to recognize expressions in this subset and specify the transformation rules.

Another contribution is to show how the existing translation between UML+OCL and Alloy [2, 1] can be extended to target the identified Alloy subset. In particular, we show how to handle correctly OCL pre- and post-conditions occurring in method specification. This allows us to fully support the aforementioned roundtrip scenarios. Our approach is illustrated with a couple of case studies.

The rest of the document is structured as follows. Next section introduces preliminary concepts related to Alloy. We then formally present, in Sections 3 and 4, respectively, the source Alloy and target UML+OCL meta-models accepted by our translation. The relationship between Alloy and CDs is presented in Section 5. The transformation from Alloy to OCL is presented in Section 6, and the opposite translation, from OCL to Alloy, in Section 7. Section 8 presents some of the case studies that were developed to validate the proposed transformation. The related work is discussed in Section 9 and the last section presents the conclusions and future work.

2 Alloy

Alloy is a formal language based on first-order relational logic [22]. It is supported by the Alloy Analyzer tool, that uses SAT solvers to enable automatic model verification, bounded by a user specified scope. An Alloy specification consists of a module with a set of imports and zero or more paragraphs. 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 the 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 latter, 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 [18]. Without loss of generality, we will also assume the distinguished state signature to be denoted as `Time`. 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, in functions 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 [22]. Besides the mandatory `Time` signature, the model introduces two top level signatures, namely `Book` to model address books and `Target` to model their content. A target can either be a `Name` or an `Addr`. Some addresses are emails. This set is modeled by signature `email`, which is a subset of `Addr`. Names can be of two types: `Alias` or `Group`. Such types were declared in the enumeration signature `Type`, which is just a shorthand for the following alternative declaration:

```
abstract sig Type { }
one sig Alias, Group extends Type { }
```

The multiplicity keyword `one` forces `Alias` and `Group` to be singletons.

Mutable fields must specify `Time` as the last column. In this model we have two mutable fields: `names` that maps books to the addressed names, and `addr` that for each book, maps names to the respective targets. The model also declares one immutable field, namely `type`, that specifies the type of each name. Field declarations can also introduce multiplicity constraints: for example, `type` maps a name to exactly `one` type, while `names` maps each book to `some` names.

In Alloy *everything is a relation*. For example, sets and signatures are unary relations and variables are just (unary) singleton relations. As such, relational operators such as composition can be used for various purposes. For example, in the first fact the expression `b.addr.t` denotes the value of the relation `addr` of book `b` at instant `t`. Note that the relational expression `b.addr.t` is contained in the cartesian product `Name -> Target`. If we compose it with the `Target` set, we get all names in the domain of that relation. As such, the first fact is an invariant that restricts all names in the second

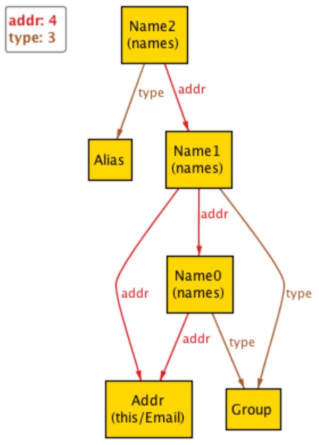

Fig. 2 An instance of an address book

column of the `addr` field of a book to be registered in the respective `names` field. The second fact also ensures a similar referential integrity, namely that all names in the third column of the `addr` field are registered in the address book (the operator `&` denotes relation intersection). The third fact imposes that all names registered in a book have some address. The fourth fact uses the transitive closure operator `^` to ensure that the `addr` relation is acyclic. The fifth fact limits the addresses of aliases to at most one target. Finally, the last fact imposes that names have at most one email address (operator `:>` restricts the range of a relation to a given set). The first `run` commands attempts to verify that the model is consistent, by requesting the Alloy Analyzer to search for a model instance containing exactly one address book that satisfies the specified invariants. Figure 2 presents one such instance, projected over the single book and time atoms specified in the scope.

Predicate `add` models an operation that adds a new target to a name in an address book. 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` is a pre-condition that requires names to be registered before adding a new target. Otherwise we have post-conditions. For example, the post-condition `b.addr.t' = b.addr.t + n->a` states that, after executing operation `add`, relation `b.addr` should have an additional tuple: operator `+` denotes relation union and the cartesian product `n->a` between singletons `n` and `a` builds a singleton binary relation, i.e, a tuple. The last post-condition establishes the frame of the operation, stating that the relation `names` does not change its value. Since Alloy has no special syntax to model operations, frame conditions must be stated explicitly. The second `run` command verifies that the `add` operation is consistent.

```
module AddressBook

sig Time { }

abstract sig Target { }

sig Addr extends Target { }
sig email in Addr { }

sig Name extends Target { type : one Type }
enum Type { Alias, Group }

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

fact { all t : Time | all b : Book | b.addr.t.Target in b.names.t }
fact { all t : Time | all b : Book | (Name.(b.addr.t) & Name) in b.names.t }
fact { all t : Time | all b : Book | all n : b.names.t | some n.(b.addr.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 n : type.Alias | lone n.(b.addr.t) }
fact { all t : Time | all b : Book | all n : Name | lone n.(b.addr.t :> email) }

run { } for 4 but exactly 1 Book, exactly 1 Time

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

run add for 3 but 1 Book, 2 Time

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

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

Fig. 1 An address book specification in Alloy.

Function `lookup` models a query that returns the set of addresses of a given name. Again, transitive closure is used to recursively traverse relation `addr`. Finally, a `check` command is used to verify that `lookup` always returns some address for every name registered in an address book. Since no counterexamples are produced, this assertion holds for all instances within the specified scope.

3 Characterizing source Alloy models

The Alloy subset accepted by our model transformation is defined in Figure 3. This subset restricts models to conform to the local state idiom, informally introduced in Section 2. In particular, the model must declare a distinguished state signature denoted `Time`. The last column of a mutable relation declaration must be `Time`. If `Time` is omitted then the relation is immutable. Mutable subsets of a signature A can be declared using a binary relation of type A `->` `Time`, that associates each $a \in A$ with the set of `Time` points where such a belongs to the set. For example, to model a (mutable) subset of `valid` addresses in our example we could change the `Addr` signature declaration as follows:

```
sig Addr extends Target {
  valid : set Time
}
```

On the other hand, to model immutable subsets we rely on signature inclusion. That is the case, for example, of the `email` subset of `Addr`. We assume that signature inclusion is only used for this particular purpose, and as such it should not declare any fields. Moreover, signatures declared with inclusion cannot be further extended or include other signatures. Support for arbitrary signature inclusion is not desirable, since it is not possible to specify using just CDs that a class is a non-disjoint subset of another class.

We capture such restrictions in the grammar by partitioning the name space of identifiers into disjoint sets, and by placing subset signatures identifiers in the namespace *irelId* of immutable relation identifiers, instead of *sigId* which denotes the set of signature identifiers. Other namespaces are: *relId* for mutable relation identifiers, *varId* for variable identifiers, *constId* for singleton signatures that extend enumerations, *funId* for function identifiers, and *predId* for predicate identifiers. The following convention will be followed for naming variables denoting the various grammar elements: x, y, z for variable identifiers (*varId*); A, B, C for signature identifiers (*sigId*) and types in general; K for singleton signature identifier (*constId*) R, S, T for mutable relation and set identifiers (*relId*); U for immutable relation and set identifier (*irelId*); ϕ, ψ, φ for formulas (*form*); $\varPhi, \varPsi, \varUpsilon$ for relational expressions (*expr*); and α, β, γ for integer expressions (*intExpr*).

Concerning operation and query modeling, all predicates must have `t` and `t'` as parameters, while functions have only `t`. Note that `Time`, `t`, and `t'` are treated as reserved keywords.

Besides these structural restrictions, the syntax of formulas is further restricted to ensure a sound operational semantics [19]. Namely, 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 presentation of the translation, we ensure this restriction by forcing mutable relation identifiers to be composed with either `t` or `t'`. However, our tool accepts a more relaxed syntax, where each occurrence of a mutable relation identifier is required to be a subterm of either $\varPhi$`.t` or $\varPhi$`.t'`, where $\varPhi$ denotes a relational expression without time variables. For example, given relations R `: ` A `-> Time` and S `: ` B `-> Time`, the expression `(`R`+`S`).t` is state-bound and would be accepted.

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 [30] and the OCL metamodel [31]:

– OCL does not allow the specification of arbitrary temporal formulas, and thus all facts must be invariants. To ensure this, facts are restricted to formulas without any time variables (that is, referring only to immutable fields), or must be of the form `all t:Time |` ϕ, with `t` the only time variable that occurs in formula ϕ.
– Field declarations must refer to 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 [9]. 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.
– 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.
– 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 3 also includes some additional minor restrictions, that do not limit the expressivity of the language, but whose inclusion would unnecessarily complicate the translation. In particular, multiplicity constraints can only occur in the last column (not counting the optional `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 ends. Multiplicities are just syntactic sugar to normal facts anyway, and thus additional multiplicities can be easily supported with a refactoring. Consider our running example. If we wanted to restrict `addr` to have at most one `Name` associated with each `Target` in each `Book`, it could have been declared as follows in standard Alloy:

```
sig Book {
  addr: Name lone -> Target set -> Time
}
```

Instead we require the following equivalent fact:

```
fact { all t : Time | all b : Book |
  all x : Target | lone b.(addr.t).x
}
```

Moreover, we do not allow signature facts, nor multiplicities in signature declaration. Again, these restrictions can easily be lifted by means of trivial refactorings.

4 Characterizing target UML+OCL models

The CDs produced and accepted by our transformation are restricted as follows:

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

Fig. 3 Subset of Alloy accepted by the transformation.

- Model types can only be classes, enumeration classes, or sets of these. Classes can optionally be abstract.
- Multiple inheritance is not supported, since it has no counterpart in Alloy strucutural features. For similar reasons, property and operation overriding is also not supported.
- The multiplicity of association ends is restricted as follows: the lower bound can either be `0` or `1`, and the upper bound `1` or `*`.
- Associations can optionally be qualified by classes or enumeration classes.
- Class atributes can also have the type `Boolean`. No other primitive type is currently supported.
- Currently, the only supported attribute of properties is `readOnly`, to distinguish between mutable and immutable properties.
- The type of an operation parameter must either be a class or an enumeration class. The return type of a query must be a set and the query itself must be side-effect free (i.e, the attribute `isQuery` is set to true). Non-query operations cannot have return values.

Likewise to [2,1], among popular UML elements not currently supported we have association classes, interfaces, and aggregation and composition associations. Given our intended usage scenarios we aim at producing human readable models. As such, our goal was to identify a subset of CDs features that have a direct counterpart in Alloy structural features. For example, it is well known that multiple inheritance can be modeled in Alloy using additional fields and facts [23], or a deep embedding strategy [24], but the resulting models would be rather cumbersome and difficult to understand. However, some of these restrictions (namely the restriction on multiplicities of association ends) can currently be supported by means of trivial refactorings to OCL invariants.

Concerning OCL, the subset supported by our transformation is presented in Figure 4. The supported constraints are class `invariants`, specification of a query result using `body`, and specification of `pre` and `post` conditions of non-query operations. Concerning OCL formulas and expressions, we significantly depart from the previous translation between OCL and Alloy [2,1] in a key issue concerning OCL semantics. When an associ-

```
     package ::= package packageId constraint* endpackage
   contraint ::= context classId inv |
                 context classId::queryId([decl(,decl)*]):Set(classId) body |
                 context classId::opId([decl(,decl)*]) (pre | post)*
         inv ::= inv: form
        body ::= body: setExpr
         pre ::= pre: form
        post ::= post: form
        decl ::= varId:classId
        form ::= true | false | not form | form and form | form or form | form implies form | if form then form else form endif |
                 object = object | intExpr compOp intExpr | setOrObject->forAll(varId | form) | setOrObject->exists(varId | form) |
                 setOrObject->isEmpty() | setOrObject->includes(object) | object.propId[@pre] | object.oclIsKindOf(classId)
      compOp ::= = | < | > | <= | >=
      object ::= self | varId | classId::constId | object.oclAsType(classId)
         ref ::= object.propId[@pre][[object(,object)*]]
     setExpr ::= classId.allInstances() | setOrObject->union(setExpr) | setOrObject->intersection(setExpr) |
                 setOrObject->select(varId | form) | setOrObject->reject(varId | form) | setOrObject->closure(varId | setExpr)
                 setOrObject->asSet() | object.queryId[@pre]([object(,object)*])
 setOrObject ::= object | ref | setExpr
     intExpr ::= integer | intExpr + intExpr | intExpr - intExpr | setOrObject->size() | setOrObject->collect(varId | intExpr)->sum()
```

Fig. 4 Subset of OCL targeted by the transformation.

ation has multiplicity `0..1`, OCL navigation may yield an undefined value, leading to a three valued logic semantics. Since Alloy's logic is the standard two-valued one, Anastasakis et al. [2,1] address this issue by assuming that an OCL undefined expression is equivalent to an empty set, trading off a semantics preserving transformation for a more flexible syntax. On the other hand, we prefer to restrict the syntax to an OCL subset that avoids such undefined expressions, and thus achieve a semantics preserving transformation. In particular, we require all navigations to be handled as sets, by constraining them to be followed by a set operation or property (signaled by the operator `->`). Note that this does not restrict the expressivity of the supported OCL subset when compared to [2,1]; at most it can lead to a more cumbersome usage. In fact, we support a more expressive syntax, since, besides the usual set operations and logical connectives, we also support the `closure` operation recently added to OCL 2.3 [31].

5 Relationship between Alloy and UML class diagrams

The relationship between CDs and Alloy declarations is straightforward, as noticed in [2,1,34,25]: in general, classes correspond to signatures (preserving the inheritance relation), associations and attributes to fields, and methods to predicates and functions.

We consider the same relationships in our translations, but with the novelty that we handle both mutable and immutable fields, and some fields may lead to non-binary associations. As seen in Figure 3, a field can be of three kinds:

- An immutable relation with type U `:` A_1 `-> ... ->` m A_n, where m is a multiplicity constraint, to be mapped to a `readOnly` qualified association U between A_1 and A_n, with $A_2, \ldots, A_{n-1}$ as qualifiers.
- A mutable relation with type R `:` A `-> set Time`, denoting an mutable subset R of A, to be mapped to an attribute R of class A with type `Boolean`.
- A mutable relation with type R `:` A_1 `-> ... ->` A_n m `-> Time`, to be mapped to a qualified association R between A_1 and A_n, with $A_2, \ldots, A_{n-1}$ as qualifiers.

For associations, the multiplicity at the end 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 signature to class mapping also has a couple of exceptions:

- A signatures declared as `sig` U `in` A `{ }`, denoting an immutable subset U of A, will be mapped to a `readOnly` attribute U of class A with type `Boolean`.

– An enumeration signature `enum` A `{` K_1`,...,`K_n `}` will be mapped to an enumerated data type declaration: A will be mapped to an enumeration class and K_1, ..., K_n to enumeration literals of that class.

6 From Alloy to OCL

The model transformation from the Alloy subset described in Section 3 to the OCL subset defined in Section 4 will be formalized using an embedding function $[\![\cdot]\!]$. To simplify the presentation, this function will accept and produce concrete syntax.

6.1 Typing issues

We will assume the source Alloy model to be well-typed, according to the typing system described in [9]. 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 Φ`.`Ψ is well-defined for any relational expressions Φ`:` A `->` B and Ψ`:` 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). For each non-abstract signature A that is extended, a special atomic signature A`$` is used to denote the set of atoms not contained in any of its extensions. The type inference rules ensure that all the tuples in the type relation have the same arity. Given a relational expression Φ of arity $|\Phi|$, we will denote the type of its n-th column as Φ^n (assuming $0 < n \leqslant |\Phi|$). The type of a column is guaranteed to be a set of atomic signatures. For example, consider the following Alloy model:

```
sig A {s : set B}
sig B extends A {}
```

The type of relation `s` is $\{\langle$`A$`,`B`$\rangle, \langle$`B`,`B`$\rangle\}$, meaning that it either contains tuples whose first component is in `A$` and second component is in `B` or tuples whose first component is in `B` and second component is also in `B`. In this case $\mathtt{s}^1$ is {`A$`,`B`} and $\mathtt{s}^2$ is {`B`}.

We will often need to quantify over such arbitrary type columns. To do so in OCL, the quantification must be reduced to iterations over concrete classes. As such, the shorthand notation presented in Figure 5 will be used, where $\{A_1, \ldots, A_n\}$ denotes a set of atomic signatures.

To avoid formula explosion, if a subset of a type exactly matches the type of a signature we quantify instead over the respective class. For example,

```
{A$,B}.allInstances()->forAll(...)
```

would be translated to the expected

```
A.allInstances()->forAll(...)
```

since the type of signature `A` is {`A$`,`B`}. If after such merging, an atomic signature denoting a remainder type (such as `A$`) is still present, the set of its instances must be explicitly computed in OCL, since it does not correspond to an existing class. This can be easily done by subtracting from the parent class all instances of its extensions. For example

```
A$.allInstances()
```

would be translated as

```
A.allInstances()->reject(a | a.oclIsKindOf(B))
```

Occasionally, we will also need to determine the parent of a given signature: given signature A we will denote its parent as $\overline{A}$, when defined. Technically, the need to determine the type of an expression (or the parent of a given signature) makes it necessary to parameterize the embedding function with information about the meta-model (namely, the `extends` relation). To simplify the presentation, we will leave this parameter implicit.

6.2 Module, Fact, Function and Predicate Declarations

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

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

Figure 6 details the transformations of fact, function and predicate declarations. Signature and enumeration declarations are ignored in the OCL generation, and are only used in the CD generation detailed in the previous section.

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.

$\{A_1,\ldots,A_n\}$`.allInstances()->forAll(`x`|`ϕ`)` $\equiv$
A_1`.allInstances()->forAll(`x`|`ϕ`) and ... and `A_n`.allInstances()->forAll(`x`|`ϕ`)`
$\{A_1,\ldots,A_n\}$`.allInstances()->exists(`x`|`ϕ`)` $\equiv$
A_1`.allInstances()->exists(`x`|`ϕ`) or ... or `A_n`.allInstances()->exists(`x`|`ϕ`)`
$\{A_1,\ldots,A_n\}$`.allInstances()->select(`x`|`ϕ`)` $\equiv$
A_1`.allInstances()->select(`x`|`ϕ`)->union(...->union(`A_n`.allInstances()->select(`x`|`ϕ`))...)`
$\{A_1,\ldots,A_n\}$`.allInstances()->collect(`x`|`ϕ`)->sum()` $\equiv$
A_1`.allInstances()->collect(`x`|`ϕ`)->sum() + ... + `A_n`.allInstances()->collect(`x`|`ϕ`)->sum()`
Φ`.isKindOf(`$\{A_1,\ldots,A_n\}$`)` $\equiv \Phi$`.isKindOf(`A_1`) or ... or `Φ`.isKindOf(`A_n`)`

Fig. 5 OCL shorthand notation for iteration over sets of classes.

$[\![$`fact {all t:Time | all `x`:`A` | `ϕ`}`$]\!] \equiv$
    `context` A
        `inv:` $[\![\phi]\!]_x$
$[\![$`fact {all `x`:`A` | `ϕ`}`$]\!] \equiv$
    `context` A
        `inv:` $[\![\phi]\!]_x$
$[\![$`fun `f`[`x_1`:`A_1`,...,`x_n`:`A_n`,t:Time] : set `B` { `Φ` }`$]\!] \equiv$
    `context` A_1`::`f`(`x_2`:`A_2`,...,`x_n`:`A_n`):Set(`B`)`
        `body:` B`.allInstances()->select(`y`|`$[\![\langle y\rangle \in \Phi]\!]_{x_1}$`)`
$[\![$`pred `f`[`x_1`:`A_1`,...,`x_n`:`A_n`,t,t':Time] { `$\phi_1 \ldots \phi_m$` }`$]\!] \equiv$
    `context` A_1`::`f`(`x_2`:`A_2`,...,`x_n`:`A_n`)`
        `pre:` $[\![\phi_1]\!]_{x_1}$ if `t'` does not occur in ϕ_1
        `post:` $[\![\phi_1]\!]'_{x_1}$ otherwise
        ...
        `pre:` $[\![\phi_m]\!]_{x_1}$ if `t'` does not occur in ϕ_m
        `post:` $[\![\phi_m]\!]'_{x_1}$ otherwise

Fig. 6 Translation of Alloy fact, function and predicate declarations.

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

6.3 Formulas

The translation of formulas is presented in Figure 7. We omit the definition of $[\![\cdot]\!]'$ because for formulas it is identical - it will only differ when applied to relational expressions. We also omit the definition for the (alternative) textual versions of logical operators and the infix negated forms. Most logic operators have a direct counterpart in OCL and can thus be trivially translated. OCL does not support directly the non-standard quantifiers `no` and `one`, but they can be encoded, for instance, using equivalent formulas that check the cardinality of the subset of the type satisfying the quantified formula.

The trickiest part of the translation concerns the atomic formulas Φ `in` Ψ, where Φ and Ψ 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 Φ will be translated to a formula that checks if a tuple $\langle y_1,\ldots,y_{|\Phi|}\rangle$ belongs to the denoted relation. The inclusion Φ `in` Ψ can thus be translated to a formula that checks if all tuples of the appropriate type that belong to Φ also belong to Ψ. Note that the type system ensures that the arity of Φ and Ψ are the same.

6.4 Expressions

The translation of relational expressions is presented in Figure 8. As explained above, this translation 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.
– Testing if a unary tuple is a member of a signature depends on the kind of signature declaration: if it is a subset signature, we just check the value of the generated boolean attribute; if it is a singleton signature that extends an enumeration we check if it

$$
\begin{aligned}
\llbracket \varPhi\ \texttt{in}\ \varPsi \rrbracket_x &\equiv \varPhi^1\texttt{.allInstances()->forAll(}y_1\,\texttt{|}\ \ldots\\
&\qquad \varPhi^{|\varPhi|}\texttt{.allInstances()->forAll(}y_{|\varPhi|}\,\texttt{|}\ \llbracket \langle y_1,\ldots,y_{|\varPhi|}\rangle \in \varPhi \rrbracket_x\ \texttt{implies}\ \llbracket \langle y_1,\ldots,y_{|\varPhi|}\rangle \in \varPsi \rrbracket_x\texttt{)}\ \ldots\texttt{)}\\
\llbracket \varPhi\ \texttt{=}\ \varPsi \rrbracket_x &\equiv \llbracket \varPhi\ \texttt{in}\ \varPsi \rrbracket_x\ \texttt{and}\ \llbracket \varPsi\ \texttt{in}\ \varPhi \rrbracket_x\\
\llbracket \phi\ \texttt{\&\&}\ \psi \rrbracket_x &\equiv \llbracket \phi \rrbracket_x\ \texttt{and}\ \llbracket \psi \rrbracket_x\\
\llbracket \phi\ \texttt{||}\ \psi \rrbracket_x &\equiv \llbracket \phi \rrbracket_x\ \texttt{or}\ \llbracket \psi \rrbracket_x\\
\llbracket \phi\ \texttt{<=>}\ \psi \rrbracket_x &\equiv \llbracket \phi\ \texttt{=>}\ \psi \rrbracket_x\ \texttt{and}\ \llbracket \psi\ \texttt{=>}\ \phi \rrbracket_x\\
\llbracket \phi\ \texttt{=>}\ \psi \rrbracket_x &\equiv \llbracket \phi \rrbracket_x\ \texttt{implies}\ \llbracket \psi \rrbracket_x\\
\llbracket \phi\ \texttt{=>}\ \psi\texttt{,}\varphi \rrbracket_x &\equiv \texttt{if}\ \llbracket \phi \rrbracket_x\ \texttt{then}\ \llbracket \psi \rrbracket_x\ \texttt{else}\ \llbracket \varphi \rrbracket_x\ \texttt{endif}\\
\llbracket \texttt{!}\phi \rrbracket_x &\equiv \texttt{not}\ \llbracket \phi \rrbracket_x\\
\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\\
\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\\
\llbracket \alpha\ \texttt{>=}\ \beta \rrbracket_x &\equiv \llbracket \alpha \rrbracket_x\ \texttt{>=}\ \llbracket \beta \rrbracket_x\\
\llbracket \texttt{no}\ \varPhi \rrbracket_x &\equiv \llbracket \texttt{\#}\varPhi\ \texttt{= 0} \rrbracket_x\\
\llbracket \texttt{lone}\ \varPhi \rrbracket_x &\equiv \llbracket \texttt{\#}\varPhi\ \texttt{=< 1} \rrbracket_x\\
\llbracket \texttt{one}\ \varPhi \rrbracket_x &\equiv \llbracket \texttt{\#}\varPhi\ \texttt{= 1} \rrbracket_x\\
\llbracket \texttt{some}\ \varPhi \rrbracket_x &\equiv \llbracket \texttt{\#}\varPhi\ \texttt{>= 1} \rrbracket_x\\
\llbracket \texttt{all}\ y\texttt{:}\varPhi\ \texttt{|}\ \phi \rrbracket_x &\equiv \varPhi^1\texttt{.allInstances()->forAll(}y\texttt{|}\llbracket \langle y\rangle \in \varPhi \rrbracket\ \texttt{implies}\ \llbracket \phi \rrbracket_x\texttt{)}\\
\llbracket \texttt{some}\ y\texttt{:}\varPhi\ \texttt{|}\ \phi \rrbracket_x &\equiv \varPhi^1\texttt{.allInstances()->exists(}y\texttt{|}\llbracket \langle y\rangle \in \varPhi \rrbracket\ \texttt{and}\ \llbracket \phi \rrbracket_x\texttt{)}\\
\llbracket \texttt{no}\ y\texttt{:}\varPhi\ \texttt{|}\ \phi \rrbracket_x &\equiv \varPhi^1\texttt{.allInstances()->select(}y\texttt{|}\llbracket \langle y\rangle \in \varPhi \rrbracket\ \texttt{and}\ \llbracket \phi \rrbracket_x\texttt{)->isEmpty()}\\
\llbracket \texttt{lone}\ y\texttt{:}\varPhi\ \texttt{|}\ \phi \rrbracket_x &\equiv \varPhi^1\texttt{.allInstances()->select(}y\texttt{|}\llbracket \langle y\rangle \in \varPhi \rrbracket\ \texttt{and}\ \llbracket \phi \rrbracket_x\texttt{)->size() <= 1}\\
\llbracket \texttt{one}\ y\texttt{:}\varPhi\ \texttt{|}\ \phi \rrbracket_x &\equiv \varPhi^1\texttt{.allInstances()->select(}y\texttt{|}\llbracket \langle y\rangle \in \varPhi \rrbracket\ \texttt{and}\ \llbracket \phi \rrbracket_x\texttt{)->size() = 1}
\end{aligned}
$$

Fig. 7 Translation of Alloy formulas.

is equal to the respective enumeration literal; otherwise, we test if it is one of the instances of the respective class.

- To test if an unary tuple belongs to R.t, which is only possible if R denotes a mutable set, we just check the value of the generated boolean attribute.
- To test if a non unary tuple belongs to R (when R is immutable) or $R.t$ (when R is mutable) we just navigate the respective qualified association R.
- The semantics of the relational composition $\varPhi$.$\varPsi$ leads to a new existential quantifier over the mediating type. We quantify over $\varPhi^{|\varPhi|} \cap \varPsi^1$ because the composition only succeeds for values belonging to the intersection of both types. This optimization reduces the number of quantifiers in the output formula.
- Testing if $\langle y_1, \ldots, y_n\rangle$ is included in the relational overriding $\varPhi$ ++ $\varPsi$ is reduced to a membership test over $\varPsi$ if $\langle y_1, \ldots, y_{n-1}\rangle$ belongs to its domain; otherwise a membership test over $\varPhi$ is generated.
- Translation of transitive closure can be done with the OCL `closure` operation, that iteratively accumulates the set denoted by the enclosed expression until a fixed point is reached.
- In a relation defined by the comprehension $\{z_1 \texttt{:} \varPhi_1\texttt{,}\ \ldots\texttt{,}\ z_n \texttt{:} \varPhi_n\ \texttt{|}\ \phi\}$, the membership test is translated by just applying the predicate ϕ to the tuple variables $y_1, \ldots, y_n$ instead of $z_1, \ldots, z_n$ and checking if the tuple variables are contained in the respective sets.

The translation of relational expressions occurring in post-conditions is almost identical, with the exception of the rules presented in Figure 9, where relation identifiers within scope `t` are evaluated in the pre-state.

Figure 10 presents the translation of Alloy integer expressions to OCL. Alloy expression #$\varPhi$ computes the size of a relational expression $\varPhi$ of arbitrary arity. In OCL we can only compute the size of sets, so we reduce such computation to sets by resorting to the following equivalence, that holds when $|\varPhi| > 1$.

$$\texttt{\#}\varPhi = \sum_{y \in \varPhi^1} \texttt{\#}(y.\varPhi)$$

6.5 Simplifications and casts

The blind application of the above translation rules usually results in obfuscated OCL specifications, mainly

$$\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 y\texttt{.oclIsKindOf(}A\texttt{)}\\
\llbracket\langle y\rangle \in K\rrbracket_x &\equiv y\texttt{=}\overline{K}\texttt{::}K\\
\llbracket\langle y_1,\dots,y_n\rangle \in U\rrbracket_x &\equiv \begin{array}{ll} y_1\texttt{.}U\texttt{[}y_2\texttt{,}\dots\texttt{,}y_{n-1}\texttt{]->includes(}y_n\texttt{)} & \text{if } n > 1 \\ y_1\texttt{.}U & \text{otherwise}\end{array}\\
\llbracket\langle y_1,\dots,y_n\rangle \in R\texttt{.t}\rrbracket_x &\equiv \begin{array}{ll} y_1\texttt{.}R\texttt{[}y_2\texttt{,}\dots\texttt{,}y_{n-1}\texttt{]->includes(}y_n\texttt{)} & \text{if } n > 1 \\ y_1\texttt{.}R & \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,\dots,y_n\rangle \in \Phi\texttt{.}\Psi\rrbracket_x &\equiv (\Phi^{|\Phi|} \cap \Psi^1)\texttt{.allInstances()->exists(}y\texttt{|}\ \llbracket\langle y_1,\dots,y_{|\Phi|-1},y\rangle \in \Phi\rrbracket_x \texttt{ and } \llbracket\langle y,y_{|\Phi|},\dots,y_n\rangle \in \Psi\rrbracket_x\texttt{)}\\
\llbracket\langle y_1,\dots,y_n\rangle \in \Phi \texttt{ + } \Psi\rrbracket_x &\equiv \llbracket\langle y_1,\dots,y_n\rangle \in \Phi\rrbracket_x \texttt{ or } \llbracket\langle y_1,\dots,y_n\rangle \in \Psi\rrbracket_x\\
\llbracket\langle y_1,\dots,y_n\rangle \in \Phi \texttt{ - } \Psi\rrbracket_x &\equiv \llbracket\langle y_1,\dots,y_n\rangle \in \Phi\rrbracket_x \texttt{ and (not } \llbracket\langle y_1,\dots,y_n\rangle \in \Psi\rrbracket_x\texttt{)}\\
\llbracket\langle y_1,\dots,y_n\rangle \in \Phi \texttt{ \& } \Psi\rrbracket_x &\equiv \llbracket\langle y_1,\dots,y_n\rangle \in \Phi\rrbracket_x \texttt{ and } \llbracket\langle y_1,\dots,y_n\rangle \in \Psi\rrbracket_x\\
\llbracket\langle y_1,\dots,y_n\rangle \in \Phi \texttt{ <: } \Psi\rrbracket_x &\equiv \llbracket\langle y_1\rangle \in \Phi\rrbracket_x \texttt{ and } \llbracket\langle y_1,\dots,y_n\rangle \in \Psi\rrbracket_x\\
\llbracket\langle y_1,\dots,y_n\rangle \in \Phi \texttt{ :> } \Psi\rrbracket_x &\equiv \llbracket\langle y_1,\dots,y_n\rangle \in \Phi\rrbracket_x \texttt{ and } \llbracket\langle y_n\rangle \in \Psi\rrbracket_x\\
\llbracket\langle y_1,\dots,y_n\rangle \in \Phi \texttt{ -> } \Psi\rrbracket_x &\equiv \llbracket\langle y_1,\dots,y_{|\Phi|}\rangle \in \Phi\rrbracket_x \texttt{ and } \llbracket\langle y_{|\Phi|+1},\dots,y_n\rangle \in \Psi\rrbracket_x\\
\llbracket\langle y_1,\dots,y_n\rangle \in \Phi \texttt{ ++ } \Psi\rrbracket_x &\equiv \texttt{if } \llbracket\langle y_1,\dots,y_{n-1}\rangle \in \Psi\texttt{.}\Psi^n\rrbracket_x \texttt{ then } \llbracket\langle y_1,\dots,y_n\rangle \in \Psi\rrbracket_x \texttt{ else } \llbracket\langle y_1,\dots,y_n\rangle \in \Phi\rrbracket_x \texttt{ endif}\\
\llbracket\langle y_1,\dots,y_n\rangle \in \texttt{~}\Phi\rrbracket_x &\equiv \llbracket\langle y_n,\dots,y_1\rangle \in \Phi\rrbracket_x\\
\llbracket\langle y_1,y_2\rangle \in \texttt{\^{}}\Phi\rrbracket_x &\equiv y_1\texttt{->closure(}z_1\texttt{|}\Phi^2\texttt{.allInstances()->select(}z_2\texttt{|}\llbracket\langle z_1,z_2\rangle \in \Phi\rrbracket\texttt{))->includes(}y_2\texttt{)}\\
\llbracket\langle y_1,y_2\rangle \in \texttt{*}\Phi\rrbracket_x &\equiv y_1\texttt{=}y_n \texttt{ or } \llbracket\langle y_1,y_n\rangle \in \texttt{\^{}}\Phi\rrbracket_x\\
\llbracket\langle y_1,\dots,y_n\rangle \in \{z_1\texttt{:}\Phi_1\texttt{,}\dots\texttt{,}z_n\texttt{:}\Phi_n \texttt{ | } \phi\}\rrbracket_x &\equiv \llbracket\langle y_1\rangle \in \Phi_1\rrbracket \texttt{ and } \dots \texttt{ and } \llbracket\langle y_n\rangle \in \Phi_n[y_1/z_1,\dots,y_{n-1}/z_{n-1}]\rrbracket \texttt{ and } \llbracket\phi[y_1/z_1,\dots,y_n/z_n]\rrbracket_x\\
\llbracket\langle y\rangle \in f\texttt{[}y_1\texttt{,}\dots\texttt{,}y_n\texttt{,t]}\rrbracket_x &\equiv y_1\texttt{.}f\texttt{(}y_2\texttt{,}\dots\texttt{,}y_n\texttt{)->includes(}y\texttt{)}
\end{aligned}$$

Fig. 8 Translation of Alloy relational expressions.

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

Fig. 9 Translation of Alloy relational expressions in post-conditions.

$$\begin{aligned}
\llbracket n\rrbracket_x &\equiv n\\
\llbracket\texttt{\#}\Phi\rrbracket_x &\equiv \begin{array}{ll} \Phi^1\texttt{.allInstances->select(}y\texttt{|}\llbracket\langle y\rangle \in \Phi\rrbracket_x\texttt{)->size()} & \text{if } |\Phi| = 1 \\ \Phi^1\texttt{.allInstances->collect(}y\texttt{|}\llbracket\texttt{\#(}y\texttt{.}\Phi\texttt{)}\rrbracket_x\texttt{)->sum()} & \text{otherwise}\end{array}\\
\llbracket\alpha \texttt{ + } \beta\rrbracket_x &\equiv \llbracket\alpha\rrbracket_x \texttt{ + } \llbracket\beta\rrbracket_x\\
\llbracket\alpha \texttt{ - } \beta\rrbracket_x &\equiv \llbracket\alpha\rrbracket_x \texttt{ - } \llbracket\beta\rrbracket_x\\
\llbracket\texttt{sum } y\texttt{:}\Phi \texttt{ | } \alpha\rrbracket_x &\equiv \Phi^1\texttt{->select(}y \texttt{ | } \llbracket\langle y\rangle \in \Phi\rrbracket_x\texttt{)->collect(}y \texttt{ | } \llbracket\alpha\rrbracket_x\texttt{)->sum()}
\end{aligned}$$

Fig. 10 Translation of Alloy integer expressions.

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 result in order to simplify it, namely, the one point rules for eliminating quantifiers. The rewrite rules currently being applied to simplify the result are presented in Figure 11. $\textsc{fc}(\Phi)$ computes the free variables of Φ and $\phi[\Phi/x]$ is the standard capture avoiding substitution of x by Φ in ϕ.

The translation of field membership presented in Figure 8 is not always safe, and may require additional type checkings and casts. Alloy's type system is quite liberal and allows access to a field from any signature that includes the owner of the field. A reference like this would generate a type error in OCL. Consider, for example, the following Alloy model:

```
sig A { r : set A }
sig B extends A { s : set A }
fact { all a : A | r.a in s.a }
```

Translation of the fact with the aforementioned rules would yield the following OCL invariant:

```
context A
  inv: A.allInstances()->forAll(v0 |
          v0.r->includes(self) implies
          v0.s->includes(self))
```

The expression `v0.s` yields a type error, since `s` is not a property of `A`. Whenever this situation occurs we include an appropriate type-check and cast to preserve the original semantics. For example, in the expression y_1`.`R`[`y_2`,...,`y_{n-1}`]->includes(`y_n`)` of Figure 8 when the type of variable y_i is not contained in R^i we output the following OCL expression:

y_i`.oclIsKindOf(`R^i`) and`

y_1`.`R`[`y_2`,...,`y_i`.oclAsType(`R^i`),...,`y_{n-1}`]->`
    `includes(`y_n`)`

To simplify the presentation, these type-checkings and casts are not included in Figure 8 but are implemented in the translation tool. For example, the OCL invariant obtained from the above fact is:

```
context A
   inv: A.allInstances()->forAll(v0 |
          v0.r->includes(self) implies
          (v0.oclIsKindOf(B) and
           v0.oclAsType(B).s->includes(self)))
```

A similar problem occurs in closures. In Alloy we can compute the closure of an expression Φ `:` A `->` B even when B is a supertype of A. Direct translation to OCL with the `closure` operation would yield an error since the accumulated set must have the same or a subtype. In such situations we cast the initial A to a B and proceed accordingly. An example of such situation occurs in our running case study, when we compute the closure of the expression `b.addr.t` with type `Name -> Target`. The result of the translation with the respective casts can be seen in Figure 18.

7 From OCL to Alloy

In this section we show how to extend the OCL to Alloy translation previously developed by Anastasakis et al. [2,1] to handle correctly dynamic issues, by resorting to the local state idiom presented in Section 3. The translation accepts the OCL subset formalized in Section 4 and will also be formalized using an (overloaded) embedding function $[\![\cdot]\!]$ that accepts and produces concrete syntax.

The translation of an OCL package with an associated UML diagram *cd* is triggered by the following rule, where $[\![cd]\!]$ denotes the Alloy structural specification obtained from applying the rules described in Section 5 to the diagram *cd*.

$$[\![\texttt{package}\ p\ c_1 \ldots c_m\ \texttt{endpackage}]\!] \equiv$$
$$\texttt{module}\ p\ \texttt{sig Time \{\}}\ [\![cd]\!]\ [\![c_1]\!] \ldots [\![c_m]\!]$$

Again we have two slightly different versions of the translation, one to be applied to invariants, query bodies, and pre-conditions, and a primed version to be applied to post-conditions. However, we no longer need to parameterize the translation with the variable denoting the `self` object, since we will use an equally named variable in Alloy to denote it instead of generating an arbitrary fresh name. We also assume a similar naming convention for variables denoting the several grammar non-terminals, namely: x, y, z for variable identifiers (*varId*); A, B, C for class identifiers (*classId*); K for enumeration literals (*constId*) R, S, T for properties with the `readOnly` attribute set to false; U for properties with the `readOnly` attribute set to true; o for object expressions (*object*); ϕ, ψ, φ for formulas (*form*); Φ, Ψ, Υ for set or object expressions (*setOrObject*); and α, β, γ for integer expressions (*intExpr*).

The translation of OCL constraints is presented in Figure 12, and basically undoes the transformation presented in Figure 6: invariants yield facts that quantify over `Time` and the signature corresponding to the context class; queries yield functions with two extra parameters that denote the `self` object and the `Time` atom required by the local state idiom; and non-query operations yield predicates with three extra parameters to

$$
\begin{aligned}
\mathit{\Phi}\texttt{->exists(}x\texttt{ | }x\texttt{=}\mathit{\Psi}\texttt{ and }\phi\texttt{)} &\equiv \phi[\mathit{\Psi}/x],\ \text{if } y \notin \mathrm{FV}(\mathit{\Phi})\\
\mathit{\Phi}\texttt{->forAll(}x\texttt{ | }x\texttt{=}\mathit{\Psi}\texttt{ implies }\phi\texttt{)} &\equiv \phi[\mathit{\Psi}/x],\ \text{if } y \notin \mathrm{FV}(\mathit{\Phi})\\
\mathit{\Phi}\texttt{ = }\mathit{\Phi} &\equiv \texttt{true}\\
\texttt{true and }\phi &\equiv \phi\\
\phi\texttt{ and true} &\equiv \phi\\
\phi\texttt{ and }\phi &\equiv \phi\\
\texttt{false or }\phi &\equiv \phi\\
\phi\texttt{ or false} &\equiv \phi\\
\phi\texttt{ or }\phi &\equiv \phi\\
\texttt{true implies }\phi &\equiv \phi\\
\mathit{\Phi}\texttt{.forAll(}x\texttt{ | }\phi\texttt{)} &\equiv \mathit{\Phi}\texttt{.forAll(}x\texttt{ | }\phi[\texttt{true}/\mathit{\Phi}\texttt{.includes(}x\texttt{)}]\texttt{)}\\
\mathit{\Phi}\texttt{.exists(}x\texttt{ | }\phi\texttt{)} &\equiv \mathit{\Phi}\texttt{.exists(}x\texttt{ | }\phi[\texttt{true}/\mathit{\Phi}\texttt{.includes(}x\texttt{)}]\texttt{)}\\
\mathit{\Phi}\texttt{.select(}x\texttt{ | }\phi\texttt{)} &\equiv \mathit{\Phi}\texttt{.select(}x\texttt{ | }\phi[\texttt{true}/\mathit{\Phi}\texttt{.includes(}x\texttt{)}]\texttt{)}\\
\mathit{\Phi}\texttt{.collect(}x\texttt{ | }\phi\texttt{)} &\equiv \mathit{\Phi}\texttt{.collect(}x\texttt{ | }\phi[\texttt{true}/\mathit{\Phi}\texttt{.includes(}x\texttt{)}]\texttt{)}\\
A\texttt{.allInstances().forAll(}x\texttt{ | }\phi\texttt{)} &\equiv A\texttt{.allInstances().forAll(}x\texttt{ | }\phi[\texttt{true}/x\texttt{.oclIsKindOf(}A\texttt{)}]\texttt{)}\\
A\texttt{.allInstances().exists(}x\texttt{ | }\phi\texttt{)} &\equiv A\texttt{.allInstances().forAll(}x\texttt{ | }\phi[\texttt{true}/x\texttt{.oclIsKindOf(}A\texttt{)}]\texttt{)}\\
A\texttt{.allInstances().select(}x\texttt{ | }\phi\texttt{)} &\equiv A\texttt{.allInstances().forAll(}x\texttt{ | }\phi[\texttt{true}/x\texttt{.oclIsKindOf(}A\texttt{)}]\texttt{)}\\
A\texttt{.allInstances().collect(}x\texttt{ | }\phi\texttt{)} &\equiv A\texttt{.allInstances().forAll(}x\texttt{ | }\phi[\texttt{true}/x\texttt{.oclIsKindOf(}A\texttt{)}]\texttt{)}\\
A\texttt{.allInstances().select(}x\texttt{ | }\mathit{\Phi}\texttt{).includes(}x\texttt{)} &\equiv \mathit{\Phi},\ \text{if } x \notin \mathrm{FV}(\mathit{\Phi})\\
\mathit{\Phi}\texttt{->closure(}x\texttt{ | }\mathit{\Psi}\texttt{)->asSet()} &\equiv \mathit{\Phi}\texttt{->closure(}x\texttt{ | }\mathit{\Psi}\texttt{)->asSet()}\\
\mathit{\Phi}\texttt{->asSet()->size()} &\equiv \mathit{\Phi}\texttt{->size()}
\end{aligned}
$$

Fig. 11 OCL Simplification rules.

$$
\begin{aligned}
&\llbracket\texttt{context } A\ c\rrbracket \equiv\\
&\qquad \texttt{fact \{ all t:Time | all self:}A\texttt{ | }\llbracket c\rrbracket\texttt{ \}}\\
&\llbracket\texttt{context } A\texttt{::}q\texttt{(}x_1\texttt{:}A_1\texttt{,}\dots\texttt{,}x_n\texttt{:}A_n\texttt{):Set(}B\texttt{) } c\rrbracket \equiv\\
&\qquad \texttt{fun } q\texttt{[self:}A\texttt{,}x_1\texttt{:}A_1\texttt{,}\dots\texttt{,}x_n\texttt{:}A_n\texttt{,t:Time] : set } B\ \{\llbracket c\rrbracket\}\\
&\llbracket\texttt{context } A\texttt{::}m\texttt{(}x_1\texttt{:}A_1\texttt{,}\dots\texttt{,}x_n\texttt{:}A_n\texttt{) } c_1 \dots c_n\rrbracket \equiv\\
&\qquad \texttt{pred } m\texttt{[self:}A\texttt{,}x_1\texttt{:}A_1\texttt{,}\dots\texttt{,}x_n\texttt{:}A_n\texttt{,t,t':Time] } \{\llbracket c_1\rrbracket \dots \llbracket c_n\rrbracket\}
\end{aligned}
$$

$$
\begin{aligned}
\llbracket\texttt{inv: }\phi\rrbracket &\equiv \llbracket\phi\rrbracket\\
\llbracket\texttt{body: }\mathit{\Phi}\rrbracket &\equiv \llbracket\mathit{\Phi}\rrbracket\\
\llbracket\texttt{pre: }\phi\rrbracket &\equiv \llbracket\phi\rrbracket\\
\llbracket\texttt{post: }\phi\rrbracket &\equiv \llbracket\phi\rrbracket'
\end{aligned}
$$

Fig. 12 Translation of OCL contraints.

denote the `self` object and the two `Time` atoms denoting the pre- and post-state.

The translation of OCL formulas is presented in Figure 13. For most cases it is straightforward, since Alloy has direct counterparts for most logic connectives. Surprisingly, Alloy has no literals to denote the true and false values, so we choose some simple expressions to denote them, namely `no none` and `some none`. Testing if a boolean attribute R holds for some object o can be done with an inclusion test for the corresponding binary relation R projected over the current time atom.

$$
\begin{aligned}
\llbracket\texttt{true}\rrbracket &\equiv \texttt{no none}\\
\llbracket\texttt{false}\rrbracket &\equiv \texttt{some none}\\
\llbracket\texttt{not }\phi\rrbracket &\equiv \texttt{!}\llbracket\phi\rrbracket\\
\llbracket\phi\texttt{ and }\psi\rrbracket &\equiv \llbracket\phi\rrbracket\texttt{ \&\& }\llbracket\psi\rrbracket\\
\llbracket\phi\texttt{ or }\psi\rrbracket &\equiv \llbracket\phi\rrbracket\texttt{ || }\llbracket\psi\rrbracket\\
\llbracket\phi\texttt{ implies }\psi\rrbracket &\equiv \llbracket\phi\rrbracket\texttt{ => }\llbracket\psi\rrbracket\\
\llbracket\texttt{if }\phi\texttt{ then }\psi\texttt{ else }\varphi\texttt{ endif}\rrbracket &\equiv \llbracket\phi\rrbracket\texttt{ => }\llbracket\psi\rrbracket\texttt{, }\llbracket\varphi\rrbracket\\
\llbracket o_1\texttt{ = }o_2\rrbracket &\equiv \llbracket o_1\rrbracket\texttt{ = }\llbracket o_2\rrbracket\\
\llbracket\alpha\texttt{ = }\beta\rrbracket &\equiv \llbracket\alpha\rrbracket\texttt{ = }\llbracket\beta\rrbracket\\
\llbracket\alpha\texttt{ < }\beta\rrbracket &\equiv \llbracket\alpha\rrbracket\texttt{ < }\llbracket\beta\rrbracket\\
\llbracket\alpha\texttt{ > }\beta\rrbracket &\equiv \llbracket\alpha\rrbracket\texttt{ > }\llbracket\beta\rrbracket\\
\llbracket\alpha\texttt{ <= }\beta\rrbracket &\equiv \llbracket\alpha\rrbracket\texttt{ =< }\llbracket\beta\rrbracket\\
\llbracket\alpha\texttt{ >= }\beta\rrbracket &\equiv \llbracket\alpha\rrbracket\texttt{ >= }\llbracket\beta\rrbracket\\
\llbracket\mathit{\Phi}\texttt{->forAll(}y\texttt{ | }\phi\texttt{)}\rrbracket &\equiv \texttt{all }y\texttt{:}\llbracket\mathit{\Phi}\rrbracket\texttt{ | }\llbracket\phi\rrbracket\\
\llbracket\mathit{\Phi}\texttt{->exists(}y\texttt{ | }\phi\texttt{)}\rrbracket &\equiv \texttt{some }y\texttt{:}\llbracket\mathit{\Phi}\rrbracket\texttt{ | }\llbracket\phi\rrbracket\\
\llbracket\mathit{\Phi}\texttt{->isEmpty()}\rrbracket &\equiv \texttt{no }\llbracket\mathit{\Phi}\rrbracket\\
\llbracket\mathit{\Phi}\texttt{->includes(}o\texttt{)}\rrbracket &\equiv \llbracket o\rrbracket\texttt{ in }\llbracket\mathit{\Phi}\rrbracket\\
\llbracket o\texttt{.}R\rrbracket &\equiv \llbracket o\rrbracket\texttt{ in }R\texttt{.t}\\
\llbracket o\texttt{.}U\rrbracket &\equiv \llbracket o\rrbracket\texttt{ in }U\\
\llbracket o\texttt{.oclIsKindOf(}A\texttt{)}\rrbracket &\equiv \llbracket o\rrbracket\texttt{ in }A
\end{aligned}
$$

Fig. 13 Translation of OCL formulas.

$$
\begin{aligned}
\llbracket y \rrbracket &\equiv y \\
\llbracket A\texttt{::}K \rrbracket &\equiv K \\
\llbracket o\texttt{.oclAsType(}A\texttt{)} \rrbracket &\equiv \llbracket o \rrbracket \\
\llbracket o\texttt{.}R \rrbracket &\equiv \llbracket o \rrbracket\texttt{.}R\texttt{.t} \\
\llbracket o\texttt{.}R\texttt{[}o_1\texttt{,}\ldots\texttt{,}o_n\texttt{]} \rrbracket &\equiv \llbracket o_n \rrbracket\texttt{.(}\ldots\llbracket o_1 \rrbracket\texttt{.(}\llbracket o \rrbracket\texttt{.}R\texttt{.t)}\ldots\texttt{)} \\
\llbracket o\texttt{.}U \rrbracket &\equiv \llbracket o \rrbracket\texttt{.}U \\
\llbracket o\texttt{.}U\texttt{[}o_1\texttt{,}\ldots\texttt{,}o_n\texttt{]} \rrbracket &\equiv \llbracket o_n \rrbracket\texttt{.(}\ldots\llbracket o_1 \rrbracket\texttt{.(}\llbracket o \rrbracket\texttt{.}U\texttt{)}\ldots\texttt{)} \\
\llbracket A\texttt{.allInstances()} \rrbracket &\equiv A \\
\llbracket \Phi\texttt{->union(}\Psi\texttt{)} \rrbracket &\equiv \llbracket \Phi \rrbracket \texttt{ + } \llbracket \Psi \rrbracket \\
\llbracket \Phi\texttt{->intersection(}\Psi\texttt{)} \rrbracket &\equiv \llbracket \Phi \rrbracket \texttt{ \& } \llbracket \Psi \rrbracket \\
\llbracket \Phi\texttt{->select(}y \texttt{ | } \phi\texttt{)} \rrbracket &\equiv \texttt{\{}y\texttt{:}\llbracket \Phi \rrbracket \texttt{ | } \llbracket \phi \rrbracket\texttt{\}} \\
\llbracket \Phi\texttt{->reject(}y \texttt{ | } \phi\texttt{)} \rrbracket &\equiv \texttt{\{}y\texttt{:}\llbracket \Phi \rrbracket \texttt{ | !}\llbracket \phi \rrbracket\texttt{\}} \\
\llbracket \Phi\texttt{->closure(}y \texttt{ | } \Psi\texttt{)} \rrbracket &\equiv \llbracket \Phi \rrbracket\texttt{.\^{}\{}y\texttt{:univ,}z\texttt{:}\llbracket \Psi \rrbracket \texttt{ | } \llbracket \texttt{true} \rrbracket\texttt{\}} \\
\llbracket \Phi\texttt{->asSet()} \rrbracket &\equiv \llbracket \Phi \rrbracket \\
\llbracket o\texttt{.}f\texttt{(}o_1\texttt{,}\ldots\texttt{,}o_n\texttt{)} \rrbracket &\equiv f\texttt{[}\llbracket o \rrbracket\texttt{,}\llbracket o_1 \rrbracket\texttt{,}\ldots\texttt{,}\llbracket o_n \rrbracket\texttt{,t]}
\end{aligned}
$$

Fig. 14 Translation of OCL set and object expressions.

$$
\begin{aligned}
\llbracket o\texttt{.}R\texttt{@pre} \rrbracket' &\equiv \llbracket o \rrbracket'\texttt{.}R\texttt{.t} \\
\llbracket o\texttt{.}R\texttt{@pre[}o_1\texttt{,}\ldots\texttt{,}o_n\texttt{]} \rrbracket' &\equiv \llbracket o_n \rrbracket'\texttt{.(}\ldots\llbracket o_1 \rrbracket'\texttt{.(}\llbracket o \rrbracket'\texttt{.}R\texttt{.t)}\ldots\texttt{)} \\
\llbracket o\texttt{.}R \rrbracket' &\equiv \llbracket o \rrbracket'\texttt{.}R\texttt{.t'} \\
\llbracket o\texttt{.}R\texttt{[}o_1\texttt{,}\ldots\texttt{,}o_n\texttt{]} \rrbracket' &\equiv \llbracket o_n \rrbracket'\texttt{.(}\ldots\llbracket o_1 \rrbracket'\texttt{.(}\llbracket o \rrbracket'\texttt{.}R\texttt{.t')}\ldots\texttt{)} \\
\llbracket o\texttt{.}f\texttt{@pre(}o_1\texttt{,}\ldots\texttt{,}o_n\texttt{)} \rrbracket' &\equiv f\texttt{[}\llbracket o \rrbracket'\texttt{,}\llbracket o_1 \rrbracket'\texttt{,}\ldots\texttt{,}\llbracket o_n \rrbracket'\texttt{,t]} \\
\llbracket o\texttt{.}f\texttt{(}o_1\texttt{,}\ldots\texttt{,}o_n\texttt{)} \rrbracket' &\equiv f\texttt{[}\llbracket o \rrbracket\texttt{,}\llbracket o_1 \rrbracket'\texttt{,}\ldots\texttt{,}\llbracket o_n \rrbracket'\texttt{,t']}
\end{aligned}
$$

Fig. 15 Translation of OCL set and object expressions in post conditions.

$$
\begin{aligned}
\llbracket n \rrbracket &\equiv n \\
\llbracket \alpha \texttt{ + } \beta \rrbracket &\equiv \llbracket \alpha \rrbracket \texttt{ + } \llbracket \beta \rrbracket \\
\llbracket \alpha \texttt{ - } \beta \rrbracket &\equiv \llbracket \alpha \rrbracket \texttt{ - } \llbracket \beta \rrbracket \\
\llbracket \Phi\texttt{->size()} \rrbracket &\equiv \texttt{\#}\llbracket \Phi \rrbracket \\
\llbracket \Phi\texttt{->collect(}y \texttt{ | } \alpha\texttt{)->sum()} \rrbracket &\equiv \texttt{sum } y\texttt{:}\llbracket \Phi \rrbracket \texttt{ | } \llbracket \alpha \rrbracket
\end{aligned}
$$

Fig. 16 Translation of OCL integer expressions.

If it is a `readOnly` attribute U we check if the object is contained in the respective signature.

The translation of OCL set and object expressions is presented in Figure 14. Again, the translation is rather straightforward with the following exceptions:

- We assume the source model to be well typed. Given the flexibility of the Alloy type system, which allows access to a field from any atom contained in super- or sub-signature, type casts can just be ignored.
- Similarly, the `asSet()` operation can be ignored since both scalars and sets are modeled by relations in Alloy.
- Association navigation is handled by composition. If the association is not `readOnly` we also compose with the `Time` variable `t`.
- While in Alloy the closure operator can be applied to a binary relation, in OCL it accumulates a set Ψ (possibly dependent on y) over an initial set Φ until a fix-point is reached. To encode the latter using the former, we first construct a binary relation that contains all possible tuples $\langle y, \Psi(y)\rangle$ using comprehension, and then compose the initial set Φ with the closure of this relation to obtain the desired fix-point.

The translation of OCL expressions occurring in post-conditions is exactly the same as the one used in the other constraints, with the exceptions presented in Figure 15. In a post-condition a property marked with `@pre` should be evaluated in the pre-state, and thus the corresponding relation or query is accessed with the pre-state `Time` variable `t`. Otherwise, the post-state `Time` variable `t'` is used instead.

Finally, the translation of integer expressions is presented in Figure 16. The only interesting case is that of the combination of operations `collect` and `sum`: the first computes a bag with all integer expressions $\alpha(y)$ for each y in Φ, and then `sum` adds up all those values. This is exactly the same semantics of the `sum` quantifier in Alloy, that we use to translate such combination. To avoid the semantic mismatch between Alloy's wrap around semantics and OCL unbounded semantics for integers, the "Forbid Overflows" option, recently added to the version 4.2 of the Alloy Analyzer [26], should be used. This option excludes instances that would lead to arithmetic overflow, thus preventing spurious counterexamples in the analysis.

8 Deployment and Case Studies

We have implemented the proposed transformations in Haskell, and deployed them as the following command line tools:

`alloy2cd` Accepts an Alloy model conforming to the syntax of Figure 3 and produces an CD in the OMG standard XML Metadata Interchange (XMI) format.

`alloy2ocl` Accepts an Alloy model conforming to the syntax of Figure 3 and produces an OCL specification conforming to the syntax of Figure 4.

`cd2alloy` Accepts an XMI CD and produces an Alloy model containing only structural features (declarations).

`ocl2alloy` Accepts an OCL specification conforming to the syntax of Figure 4 and an associated CD and

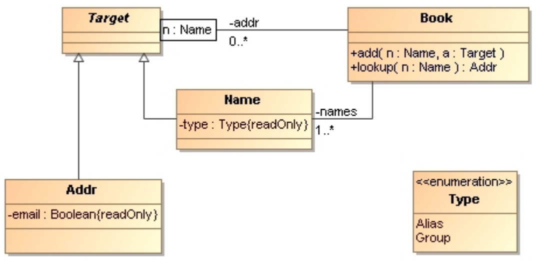

Fig. 17 UML class diagram obtained from the Alloy address book example.

produces an Alloy model conforming to the syntax of Figure 3.

These tools and the (open-source) source code are available for download at the project website:

`http://sourceforge.net/projects/alloymda`

We will now describe two simple case-studies that were used to validate our tool: the first is the address book example, and the second illustrates our bidirectional transformation, by starting with an UML+OCL specification, which is then translated for Alloy for V&V, and translated back to UML+OCL after corrections and enhancements. Other case-studies can be found in the project website. We used DresdenOCL [8] to validate the generated OCL specifications. Unfortunately, this tool still does not support qualified associations, and thus we could only validate OCL specifications with binary ones. Besides syntax and type-checking, this tools allows us to interpret OCL specifications against model instances, and to generate AspectJ code that instruments Java code with constraint checkers.

8.1 Address book

The CD generated from the address book example of Figure 1 using the `alloy2cd` tools is depicted in Figure 17. As described in Section 5, the enumeration signature `Type` was translated as an enumeration class. The ternary relation `addr` is modeled as a qualified association. Given the `one` multiplicity, relation `type` is modeled by an attribute. The subset signature `email` is also modelled by an attribute with `Boolean` type. Since both these relations are immutable, these attributes are marked as `readOnly`.

The OCL model generated using the `alloy2ocl` tool is presented in Figure 18. Except for manual indentation for better comprehension, the shown OCL is exactly the one produced by the tool. Notice the use of type-checks and casts to correctly handle the closure operation. Obviously, the `ocl2alloy` tool can be applied to the generated OCL, resulting in a syntactically more complex Alloy model, although semantically equivalent, than the one presented in Figure 1. The reason for such is that, unlike for OCL, we have still not developed simplification rules for Alloy models. We intend to do so in the near future.

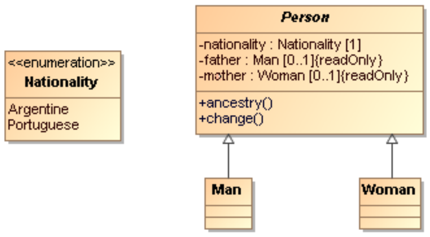
<<enumeration>>
Nationality
Argentine
Portuguese
Person
-nationality : Nationality [1]
-father : Man [0..1]{readOnly}
-mother : Woman [0..1]{readOnly}
+ancestry()
+change()
Man
Woman

Fig. 19 Simple genealogy UML diagram.

8.2 Genealogy

Consider now the CD of Figure 19. It describes a simplistic genealogy model. A `Person` can have a known biological `father` and `mother`, which are, respectively, a `Man` and a `Woman`. Moreover we model the `nationality` of a person as a singleton attribute pointing to the respective enumeration class. The `father` and `mother` properties are marked as `readOnly`. We also have a query denoted `ancestry` that is supposed to return the set of all known ancestors of a given person, and an operation named `change` that changes the nationality of a person.

This CD is complemented with the OCL specification of Figure 20. The first constraint specifies the result of the `ancestry` query using a closure, that iteratively accumulates all known fathers and mothers. Notice the usage of type checks and casts to decide if a given person is either a man or a woman. The second constraint is an invariant that states that no-one can be an ancestor of itself. The third constraint is an invariant that intends to restrict the nationality of every person to the one of the nationalities of his/her ancestors. Finally, operation `change` requires as a pre-condition that the new nationality is in one of nationalities of his/her ancestors, and the post-condition updates the nationality attribute accordingly.

Figure 22 presents the Alloy model generated with the `ocl2alloy` tool. Except for the translation of clo-

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

   context Book
      inv: Name.allInstances()->forAll(v6 |
              (Name.allInstances()->exists(v7 | self.addr[v7]->includes(v6)) implies self.names->includes(v6)))

   context Book
      inv: Name.allInstances()->forAll(n | (self.names->includes(n) implies (self.addr[n]->size() >= 1)))

   context Book
      inv: Name.allInstances()->select(n | n.oclAsType(Target)->closure(v20 | Target.allInstances()->select(v21 |
              (v20.oclIsKindOf(Name) and self.addr[v20.oclAsType(Name)]->includes(v21))))->includes(n))->isEmpty()

   context Book
      inv: Name.allInstances()->forAll(n | (n.type->includes(Type::Alias) implies (self.addr[n]->size() <= 1)))

   context Book
      inv: Name.allInstances()->forAll(n |
              (Addr.allInstances()->select(v29 | (self.addr[n]->includes(v29) and v29.email))->size() <= 1))

   context Book::add(n:Name,a:Target)
      pre: self.names->includes(n)
      pre: (not self.addr[n]->includes(a))
      post: (Name.allInstances()->forAll(v40 | Target.allInstances()->forAll(v41 |
               (self.addr[v40]->includes(v41) implies
                  (self.addr@pre[v40]->includes(v41) or ((v40 = n) and (v41 = a)))))) and
             Name.allInstances()->forAll(v46 | Target.allInstances()->forAll(v47 |
               ((self.addr@pre[v46]->includes(v47) or ((v46 = n) and (v47 = a))) implies
                  self.addr[v46]->includes(v47)))))
      post: (Name.allInstances()->forAll(v52 | (self.names->includes(v52) implies self.names@pre->includes(v52))) and
             Name.allInstances()->forAll(v57 | (self.names@pre->includes(v57) implies self.names->includes(v57))))

   context Book::lookup(n:Name):Set(Addr)
      body: n.oclAsType(Target)->closure(v64 | Target.allInstances()->select(v65 |
               (v64.oclIsKindOf(Name) and self.addr[v64.oclAsType(Name)]->includes(v65))))
endpackage
```

Fig. 18 OCL specification obtained from the Alloy address book example.

```
package Genealogy
   context Person::ancestry():Set(Person)
      body: self->closure(x | Person.allInstances()->select(y |
               y.oclIsKindOf(Man) and x.father->includes(y.oclAsType(Man)) or
               y.oclIsKindOf(Woman) and x.mother->includes(y.oclAsType(Woman))))

   context Person
      inv: not self.ancestry()->includes(self)

   context Person
      inv: Nationality.allInstances()->forAll(n | self.nationality->includes(n) implies
               self.ancestry()->exists(p | p.nationality->includes(n)))

   context Person::change(n:Nationality)
      pre: self.ancestry()->exists(p | p.nationality->includes(n))
      post: self.nationality->includes(n)
endpackage

endpackage
```

Fig. 20 OCL specification of the simple genealogy.

```
module Genealogy

sig Time {}

abstract sig Person { father : lone Man, mother : lone Woman, nationality : Nationality one -> Time }

enum Nationality {Argentine,Portuguese}

sig Man extends Person {}
sig Woman extends Person {}

fun ancestry [self : Person,t : Time] : set Person {
   self.^{x : univ,v0 : {y : Person | ((y in Man) && (y in x.father)) || ((y in Woman) && (y in x.mother))} | no none}
}
fact {all t : Time | all self : Person | self not in ancestry[self,t]}
fact {all t : Time | all self : Person | all n : Nationality |
   (n in self.(nationality.t)) => (some p : ancestry[self,t] | n in p.(nationality.t))
}
pred change [self : Person,n : Nationality,t,t' : Time] {
   some p : ancestry[self,t] | n in p.(nationality.t)
   n in self.(nationality.t')
}
```

Fig. 21 Alloy specification obtained from the simple genealogy CD and OCL.

sure, the resulting model is quite readable. We can now use the Alloy Analyzer tool to perform some V&V, for example checking the consistency of the model using a `run` command to search for a valid instance with some person. Unfortunately, no such instance exists, meaning that the original OCL specification is inconsistent. The problem is the second invariant that, to be satisfied, requires every person to have some known ancestry, which obviously cannot be satisfied by a finite instance.

In this case, it is trivial to correct the corresponding fact as presented in Figure 22. Moreover, we refine the generated Alloy model with additional mutable `wife` and `husband` relations, and new facts to ensure that these relations are symmetric and no-one marries an ancestor. We also add an operation `marry`, with a pre-condition requiring the groom to not be previously married, a post-condition that adds the bride to the `wife` relationship, and a frame-condition stating that all other men preserve their marital status.

Finally, we can translate the corrected and enhanced Alloy model back to OCL using the tool `alloy2ocl`, resulting in the specification in Figure 23. Due to the implemented simplifications, the constraints that were present in the original model are almost identical. As discussed in Section 4, the generated OCL model is a bit verbose, given our safety requirement to use the result of navigations as sets.

9 Related Work

The translation from Alloy to UML+OCL 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 [5], USE [20] or UML2Alloy [2, 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 the usage of MDA-UML tools as well as MDA-OCL tools. Most of the UML tools allow transformations from CDs 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 [7] and DresdenOCL [8].

The relationship between UML+OCL and Alloy has been extensively studied by Anastaskis et al. [2, 1], resulting in a prototype tool named UML2Alloy, that translates UML+OCL models to Alloy. Likewise to our proposal, this translation considers the basic elements of CDs: classes, attributes and associations; and excludes interfaces, dependencies and signals. Unlike our proposal, it does not support qualified associations, and CDs are restricted to binary associations. Although the authors hint that dynamic issues could be modeled in Alloy, using an idiom such as the local state one [1, page 133], the specified (and implemented) translation still does not consider them: they constraint all at-

```
module Genealogy

sig Time {}

abstract sig Person { father : lone Man, mother : lone Woman, nationality : Nationality one -> Time }

enum Nationality {Argentine,Portuguese}

sig Man extends Person { wife : Woman lone -> Time }
sig Woman extends Person { husband : Man lone -> Time }

fun ancestry [self : Person,t : Time] : set Person {
   self.^{x : univ,v0 : {y : Person | ((y in Man) && (y in x.father)) || ((y in Woman) && (y in x.mother))} | no none}
}
fact {all t : Time | all self : Person | self not in ancestry[self,t]}
fact {all t : Time | all self : Person | all n : Nationality |
   (some ancestry[self,t] and n in self.(nationality.t)) => (some p : ancestry[self,t] | n in p.(nationality.t))
}
fact {all t : Time | all p : Woman | p.(husband.t) = (wife.t).p}
fact {all t : Time | all p : Person | no (p.(wife+husband).t & ancestry[p,t])}

pred change [self : Person,n : Nationality,t,t' : Time] {
   some p : ancestry[self,t] | n in p.(nationality.t)
   n in self.(nationality.t')
}
pred marry [m : Man, w : Woman, t,t' : Time] {
   no m.wife.t
   w in m.wife.t'
   all x : Man-m | x.wife.t' = x.wife.t
}
```

Fig. 22 Corrected and enhanced Alloy specification of the simple genealogy.

```
package Genealogy
   context Person::ancestry():Set(Person)
      body: self->closure(v2 | Person.allInstances()->select(v3 |
               ((v3.oclIsKindOf(Man) and v2.father->includes(v3.oclAsType(Man))) or
                (v3.oclIsKindOf(Woman) and v2.mother->includes(v3.oclAsType(Woman))))))
   context Person
      inv: (not self.ancestry()->includes(self))
   context Person
      inv: Nationality.allInstances()->forAll(n |
              (((self.ancestry()->size() >= 1) and self.nationality->includes(n)) implies
              Person.allInstances()->exists(p | (self.ancestry()->includes(p) implies p.nationality->includes(n)))))
   context Woman
      inv: (Man.allInstances()->forAll(v23 | (self.husband->includes(v23) implies v23.wife->includes(self))) and
            Man.allInstances()->forAll(v28 | (v28.wife->includes(self) implies self.husband->includes(v28))))
   context Person
      inv: (Person.allInstances()->select(v33 |
             ((((self.oclIsKindOf(Man) and v33.oclIsKindOf(Woman)) and
                   self.oclAsType(Man).wife->includes(v33.oclAsType(Woman))) or
               ((self.oclIsKindOf(Woman) and v33.oclIsKindOf(Man)) and
                   self.oclAsType(Woman).husband->includes(v33.oclAsType(Man)))) and
             self.ancestry()->includes(v33)))->size() = 0)
   context Person::change(n:Nationality)
      pre: Person.allInstances()->exists(p | (self.ancestry()->includes(p) implies p.nationality->includes(n)))
      post: self.nationality->includes(n)
   context Man::marry(w:Woman)
      pre: (self.wife->size() = 0)
      post: self.wife->includes(w)
      post: Man.allInstances()->forAll(x | ((not (x = self)) implies
               (Woman.allInstances()->forAll(v44 | (x.wife->includes(v44) implies x.wife@pre->includes(v44))) and
                Woman.allInstances()->forAll(v49 | (x.wife@pre->includes(v49) implies x.wife->includes(v49))))))
endpackage
```

Fig. 23 OCL specification obtained from the corrected and enhanced Alloy specification of the simple genealogy.

tributes and association ends in a CD to be `readOnly`, and, unlike in our transformation, no state signature is automatically inserted in the resulting specification. In particular, operations do not receive the pre- and post-state as parameters, and thus the pre- and post-conditions used to specify operations in OCL are not translated correctly to Alloy (they conflate in the single implicit global state). To model dynamics correctly one is forced to introduce the state explicitly as a class in the CD, and to model dynamic properties explicitly as associations to that class, as suggested in [3]. Essentially, this mimics the local state idiom of Alloy in UML+OCL, disregarding the standard semantics for dynamics in the latter formalism. On the other hand, we correctly translate OCL pre- and post-conditions to Alloy specifications by resorting to the local state idiom. We also support OCL queries, by translating them to Alloy functions.

Concerning OCL logic, as discussed at the end of Section 4, we mainly differ from [2,1] in the treatment of OCL's three-valued logic: to achieve a semantics preserving transformation we restrict the supported OCL syntax to a subset that avoids undefined expressions. Another key difference is that we support the `closure` operation recently added to OCL. Note that for the common subset of OCL logic supported both by our tools and by [2,1], the result of applying the translation to Alloy is the precisely the same.

Massoni et al. also propose a UML+OCL to Alloy translation in [25]. The approach to translate classes, attributes and associations is the same as Anastaskis et al. [2,1]. Although they consider the translation of UML interfaces, they map only a small subset of UML and OCL elements. This translation is only specified and not implemented. Moreover, it translates OCL invariants but it does not handle pre- and post-conditions.

Maoz et al. [24] propose a formalization of UML class diagrams using a deep embedding to Alloy, to support UML features not directly expressible in Alloy, such as multiple inheritance or interfaces. This approach is well-suited to support automated analysis, since it enables reasoning about several CDs at once, making it possible to check, for example, that one class diagram is a refinement of another. However, unlike shallow embeddings, such as the one proposed in this paper, it generates Alloy models that are difficult to read and understand and thus it is not well suited to support our intended usage scenarios.

UML has been mapped to Alloy for model V&V of particular case-studies. For example, Georg et al. use the Alloy Analyzer for formal security evaluation in a methodology called Aspect-Oriented Risk-Driven Development (AORDD) [16], Mostefaoui and Vachon describe a proposal for deriving Alloy specifications from Aspect-UML models (a UML Profile for extending UML with Aspect-oriented concepts) [27], and Braga et al. propose an approach to translate UML models specified with OntoUML to Alloy [4]. These examples, like in [24], make evident Alloy potential for UML V&V, but they do not consider the translation of OCL specifications.

We first proposed the translation from Alloy local state idiom to UML+OCL in [14]. The present work improves that translation by allowing both mutable and inmutable fields in signature definitions, `enum` signatures, arbitrary closures of relational expressions, integer expressions $\#\Phi$ for relations Φ of arbitrary arity, among other minor improvements. We also show how the inverse translation from UML+OCL to Alloy can be defined, by improving the translation first proposed in [2,1] to handle dynamic issues and closures. This allows us to support roundtrip scenarios such as the one presented in the previous section. It also enabled us to develop a translation of UML Protocol State Machines (PSMs) to Alloy [15], that allows us to simulate and verify the consistency between UML artifacts (PSM, CD and OCL) and to perform other V&V activities, such as detect unreachable states or invalid transitions. In that work, we first use the translation described here to map a UML+OCL model to the local state idiom in Alloy, and then map a PSM (optionally, also enriched with OCL) to a trace specification that captures the allowed behavior of a component.

Appart from transitive closure, our translation from Alloy to OCL is essentially an encoding of the semantics of relational logic in terms of first-order logic. A similar technique was used recently to develop a tool for unbounded verification of Alloy models using SMT solvers [17]. A key difference to our work is the treatment of Alloy type system: since SMT does not support subtype declarations, these are encoded implicitly using membership functions and axioms to ensure the correct semantics. Another difference is the encoding of closure: since standard SMT first-order theories do not support it, an inductive definition is used instead.

Shah et al. [32] proposed a model transformation from Alloy to UML to convert model instances generated by the Alloy Analyzer to UML Object Diagrams. This translation complements the UML2Alloy tool [2, 1], which must be used before to generate an Alloy specification, and can be used to map back to UML counterexamples of UML+OCL specifications found using the Alloy Analyzer. The combination of both tools allows UML practitioners to perform V&V of UML+OCL without knowledge of Alloy and direct interaction with its Analyzer. This is very convenient for development teams only interested in using UML in the modeling phase. On the other hand, our proposed (bidirectional) transformation enables more flexible scenarios: for example, a team combining expertise in both modeling languages can start by developing a rough specification with CDs, then translate it to (the more syntax and verification friendly) Alloy for refinement (adding invariants and specification for operations), and then translate back UML+OCL to explore code generation tools.

Alloy has no specific syntactic features to model dynamic systems and to specify their properties. Dynamics can be modeled implicitly using several idioms [22], such as the local state idiom described in Section 3 (and its close variant, the global state idiom), where operations are modeled using predicates relating pre- and pos-states, or the event based idiom, where signatures are used to model events with arguments being modeled by fields. Specification and verification of invariants is quite simple with these idioms, but specification of more complex properties can be a bit cumbersome: in the local state idiom one must first model execution traces explicitly, by constraining any two consecutive states to be the pre- and post-state of one of the modeled operations, and then specify the desired property over such traces. Recently, Vakili and Day [33] have shown how branching time temporal logic can be used to simplify the later step, by proposing a library of functions that encode CTL (plus fairness conditions) temporal connectives into Alloy's relational logic. A similar approach can be followed for linear time temporal logic [6].

To allow the explicit modeling of dynamics, Frias et al. [10] introduced DynAlloy, an extension of Alloy based on dynamic logic that allows properties to be specified using actions: atomic actions can be specified using pre- and post-conditions, but can also be composed into more complex actions by means of sequencial composition, iteration, or non-deterministic choice. This extension simplifies considerably the description of operations and traces, and likewise to temporal logic, also simplifies the specification of dynamic properties over such traces. DynAlloy specifications can also be analyzed efficiently, as shown in [11]. The availability of first-class and composable actions makes DynAlloy a better target to encode the semantics (and perform analysis) of imperative-like programming languages. For example, it has been applied in the analysis of JML-annotated Java sequential programs [12,13]. Given the availability of tools to convert OCL to JML (such as [21]) it would be interesting to determine the feasibility of translating UML+OCL to Alloy by composing such tools. In particular, this would enable a proper comparison (namely, in terms of efficiency of the analysis) with the translation first proposed in [2,1] and extended in this paper.

With a similar motivation to DynAlloy, Near et al. introduced Imperative Alloy [28], an extension of the Alloy language that simplifies the specification of dynamic systems using a mix of declarative and imperative style (allowing actions to be specified with pre- and post-conditions, assignments, sequential composition, or loops). The translations proposed in this paper could trivially be adapted to target DynAlloy or Imperative Alloy. However, since OCL is a declarative specification language, where operations are specified using only pre- and post-conditions, only a subset of these languages with expressivity comparable to the local state idiom could be considered (in particular, there is no counterpart in OCL to the action composition operators). Even so, due the convenience of these extensions to reason about dynamic systems, in the near future we also intend to support them in our framework.

10 Concluding Remarks and Future Work

We have presented a bidirectional model transformation from Alloy to UML Class Diagrams annotated with OCL. Bidirectionality is achieved by means of two unidirectional transformations that are inverses of each other. The transformation from Alloy to UML+OCL is an extension of our work presented in [14], and the opposite transformation improves a previously developed one [2,1] to handle dynamic issues and closures. We have also formally characterized the Alloy local state idiom accepted and generated by the transformations, that allows us to model dynamic behavior in Alloy by introducing an explicit state signature in all mutable relations.

The model transformation from Alloy to UML+OCL raised interesting new challenges, when compared to the opposite transformation, namely: the translation of relational expressions of arbitrary arity; dealing with the idiosyncrasies of Alloy's type-system; and the encoding of closures. Both transformations are deployed as open-source tools, and can be used to support various modeling scenarios. In particular, they can be used to introduce Alloy in existing MDE software development cycles, by supporting round-trip scenarios where one starts from a high-level UML specification, translate it to Alloy for V&V, and then translate it back to UML after model refinement and debugging.

In the future we intend to enlarge the subsets of the supported Alloy and UML+OCL metamodels. The Alloy subset is already quite complete, but we still intend to research the possibility of allowing generalized usage of subset signatures to overcome the single-inheritance limitation, as discussed in [22]. The OCL subset can be easily extended to support additional set operations. We also intend to support more primitive types, namely integers, and additional collection types, such as sequences. In order to improve the readability of the Alloy generated by the `ocl2alloy` tool, we are currently developing a simplification system similar to what we have implemented for OCL. We also intend to extend our tools to, optionally, target Alloy extensions more suitable to model dynamics, namely DynAlloy [10] and Imperative Alloy [28].

Acknowledgements Part of this work was done while the first author was visiting the Software Design Group at CSAIL, MIT, USA, funded by the portuguese Foundation for Science and Technology (FCT) with a sabbatical grant with reference SFRH/BSAB/1187/2011. The second author was partially supported the portuguese National Strategy Reference Chart (QREN) project 1621 (Evolve) while visiting the High-Assurance Software Laboratory at Universidade do Minho, Portugal.

References

1. Anastasakis, K.: A model driven approach for the automated analysis of uml class diagrams. Ph.D. thesis, University of Birmingham (2009)
2. 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)
3. Bordbar, B., Anastasakis, K.: UML2ALLOY: A tool for lightweight modelling of discrete event systems. In: Proceedings of the IADIS International Conference in Applied Computing, pp. 209–216. IADIS Press (2005)
4. 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)
5. 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)
6. Cunha, A.: Bounded model checking of temporal formulas with Alloy. CoRR **abs/1207.2746** (2012)
7. 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)
8. DresdenOCL website. `http://www.dresden-ocl.org/index.php/DresdenOCL`
9. 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)
10. Frias, M.F., Galeotti, J.P., Pombo, C.L., Aguirre, N.: DynAlloy: upgrading alloy with actions. In: Proceedings of the 27th International Conference on Software Engineering, pp. 442–451. ACM (2005)
11. Frias, M.F., Pombo, C.L., Galeotti, J.P., Aguirre, N.: Efficient Analysis of DynAlloy Specifications. ACM Transactions on Software Engineering and Methodology **17**(1) (2007)
12. Galeotti, J.P., Frias, M.F.: DynAlloy as a Formal Method for the Analysis of Java Programs. In: Software Engineering Techniques: Design for Quality, *IFIP*, vol. 227, pp. 249–260. Springer-Verlag (2006)
13. Galeotti, J.P., Rosner, N., López Pombo, C.G., Frias, M.F.: Analysis of invariants for efficient bounded verification. In: Proceedings of the 19th International Symposium on Software Testing and Analysis, pp. 25–36. ACM (2010)
14. Garis, A.G., Cunha, A., Riesco, D.: Translating Alloy specifications to UML class diagrams annotated with OCL. In: Proceedings of the 9th International Conference on Software Engineering and Formal Methods, *LNCS*, vol. 7041, pp. 221–236. Springer-Verlag (2011)
15. Garis, A.G., Paiva, A.C.R., Cunha, A., Riesco, D.: Specifying UML Protocol State Machines in Alloy. In: Proceedings of the 9th International Conference on Integrated Formal Methods, *LNCS*, vol. 7321, pp. 312–326. Springer-Verlag (2012)
16. 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)
17. 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)
18. Gheyi, R., Massoni, T., Borba, P.: Formally introducing Alloy idioms. In: Proceedings of the Brazilian Symposium on Formal Methods, pp. 22–37 (2007)
19. 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)
20. 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)
21. Hamie, A.: Translating the Object Constraint Language into the Java Modelling Language. In: Proceedings of the 19th ACM Symposium on Applied Computing, pp. 1531–1535. ACM (2004)
22. Jackson, D.: Software Abstractions: Logic, Language, and Analysis, revised edn. MIT Press (2012)
23. Kim, J.S., Garlan, D.: Analyzing architectural styles with Alloy. In: Proceedings of the ISSTA 2006 Workshop on Role of Software Architecture for Testing and Analysis, pp. 70–80. ACM (2006)
24. Maoz, S., Ringert, J., Rumpe, B.: CD2Alloy: Class diagrams analysis using Alloy revisited. In: J. Whittle, T. Clark, T. Khne (eds.) Model Driven Engineering Languages and Systems, *LNCS*, vol. 6981, pp. 592–607. Springer-Verlag (2011)
25. 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)
26. Milicevic, A., Jackson, D.: Preventing arithmetic overflows in Alloy. In: Proceedings of the 3rd International Conference on Abstract State Machines, Alloy, B, VDM,

and Z, *LNCS*, vol. 7316, pp. 108–121. Springer-Verlag (2012)
27. 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)
28. Near, J.P., Jackson, D.: An imperative extension to Alloy. In: Proceedings of the Second International Conference on Abstract State Machines, Alloy, B and Z - ABZ, *LNCS*, vol. 5977, pp. 118–131. Springer-Verlag (2010)
29. OMG: MDA Guide version 1.0.1 (2003)
30. OMG: UML Superstructure, Version 2.3 (2010)
31. OMG: Object Constraint Language, Version 2.3.1 (2012)
32. 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)
33. Vakili, A., Day, N.: Temporal logic model checking in Alloy. In: Proceedings of the 3rd International Conference on Abstract State Machines, Alloy, B, VDM, and Z, *LNCS*, vol. 7316, pp. 150–163. Springer-Verlag (2012)
34. 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)