A UML Class Diagram Analyzer

Tiago Massoni, Rohit Gheyi, and Paulo Borba

Informatics Center
Federal University of Pernambuco, Brazil
{tlm,rg,phmb}@cin.ufpe.br

Abstract. Automatic analysis of UML models constrained by OCL invariants is still an open research topic. Especially for critical systems, such tool support is important for early identification of errors in modeling, before functional requirements are implemented. In this paper, we present ideas on an approach for automatic analysis of UML class diagrams, according to a precise semantics based on Alloy, a formal object-oriented modeling language. This semantics permits the use of Alloy's tool support for class diagrams, by applying constraint solving for automatically finding valid snapshots of models. This kind of automation helps the identification of inconsistencies or under-specified models of critical software, besides allowing checking of properties about these models.

1 Introduction

As in other engineering fields, modeling can be a useful activity for tackling significant problems in software development. As a de-facto standard, the Unified Modeling Language (UML) [1] plays a significant role. In particular, the development of high-quality critical systems can benefit from features offered by a standard visual modeling notation like UML, since traditional code-driven approaches are highly risky and often error-prone for such complexity. Further, business rules can be precisely expressed by Object Constraint Language (OCL) [2] invariants attached to class diagrams, enabling the specification of constraints over complex states of critical systems.

As the use of OCL for critical systems substantially grows, the lack of standardized formal semantics for the language does not stimulate the development of tools supporting analysis and verification of OCL expressions in UML models. When using class diagrams for modeling critical systems, the absence of tool support restrains the developer's task, since subtle structural modeling errors may be considerably hard to detect. For instance, inappropriate OCL invariants may turn a model over-constrained, or even inconsistent (i.e. the model allows no implementation). Likewise, models lacking important constraints may allow incorrect implementations.

In this paper, we propose an approach for automatic analysis of class diagrams, according to a precise semantics based on Alloy [3], a formal object-oriented modeling language founded on first-order logic. Alloy is suitable for object modeling, employing sets and relations as a simple semantic basis for objects and its relationships. We defined a number of mapping rules between UML/OCL and Alloy elements, resulting in equivalent Alloy models from UML class diagrams annotated with OCL invariants. By using this approach, we can leverage to UML/OCL the benefits from the powerful tool support offered by the Alloy Analyzer [4].

Alloy's simple semantics allowed powerful tool support represented by the Alloy Analyzer, which is a constraint solver that finds instances of formulae representing models, backed by an off-the-shelf SAT solver [4]. This approach allows automatic generation of snapshots satisfying model constraints, which can be significantly useful for verifying whether models are over or under-constrained. Similarly, assertions can be made against models, which are checked by exhaustive search for counterexamples refuting the assertions. Due to the undecidability of such analysis, they are parameterized by a scope (provided by the user), which assigns a bound of objects to each entity.

Since the search for a solution is limited by a scope, the absence of an instance does not automatically show that a formula is inconsistent. However, such level of automation can cover significantly more cases than any kind of testing, allowing early identification of bugs in functional requirements of critical systems. More specifically, this analysis is able to generate all valid snapshots of a model within a given scope. The Alloy language and its analysis has been successfully used for modeling critical systems, including air-traffic control [5] and a proton therapy machine [6]. We believe that similar benefits can be achieved by applying this automatic analysis to UML/OCL.

A proposed tool support is closely related [7]. The USE tool offers a useful evaluation of a user-provided snapshot of a model, checking whether this instance satisfies the model invariants. In contrast, the Alloy Analyzer offers solving, which involves searching for instances satisfying a given constraint. The latter may be more effective as an instrument for finding unexpected problems in scenarios where it is unfeasible to supply a representative set of test cases.

The remainder of this paper is organized as follows. Section 2 describes the Alloy language and the underlying tool support. In Section 3, we show the mapping rules for transforming UML class diagrams into Alloy specifications. Section 4 describes how the Alloy's tool support can be useful for analyzing UML class diagrams. Section 5 describes related work, whereas Section 6 presents our conclusions and future work.

2 Alloy

Alloy is a formal modeling language based on first-order logic, allowing specification of – primarily structural – properties in a declarative fashion. In general, models in Alloy are described at a high level of abstraction, ignoring implementation details. With Alloy, one can apply object modeling in a similar fashion to UML class diagrams, with the additional benefit of a simple semantics, allowing automatic analysis. Logical formulae can be used to enforce business rules, playing a role similar to OCL invariants. In this section, we first discuss the language, and then provide more detail on its automatic analysis.

2.1 The Alloy Language

The language is strongly typed, assuming a universe of objects partitioned into subsets, each of which associated with a basic type. An Alloy model is a sequence of paragraphs of two kinds: signatures, used for defining new types; and formula paragraphs, such as facts and predicates, used to record invariants. Analogous to classes, each signature denotes a set of objects. These objects can be mapped by the relations (associations) declared in the signatures. A signature paragraph may introduce a collection of relations.

As an example, we show an Alloy model for part of the banking system, where each bank contains a set of accounts and a set of customers. An account can only be a checking account. The next Alloy fragment declares four signatures representing system entities, along with their relations:

```
sig Bank {
  accs: set Account,
  custs: set Customer
}
sig Customer {}
sig Account {}
sig ChAcc extends Account {}
```

In the declaration of `Bank`, the `set` keyword specifies that the `accs` relation maps each object in `Bank` to a set of objects in `Account`, exactly as a 1-N association in UML. `ChAcc` denotes one kind of account. In Alloy, one signature can extend another one by establishing that the extended signature is a subset of the parent signature. For example, the set of `ChAcc` objects is a subset of the `Account` objects.

A fact is a formula paragraph. It is used to package invariants about certain sets. Differently from OCL invariants, a fact may not introduce a context for its formulae, allowing expression of global properties on models. Also, fact formulae are declared as a conjunction. The following code introduces a fact named `BankProperties`, establishing general properties about the previously declared signatures and relations.

```
fact BankProperties {
  Account = ChAcc
  all a:Account | lone a.~accs
}
```

The first formula states that every account is a checking account; the second one states that every account is related to at most one bank by `accs`. The `all` keyword is the universal quantifier. The expression `~accs` denotes the transpose of `accs`, while `lone` states that `a.~accs` yields a relation with at most one object.

The join of relations[1]. `a.~accs` is the set of all elements for which there exists an element of `a` related to it by the relation `~accs` (direct relation image). The expression yields the bank in which the `a` account is stored. Figure 1 shows how this model could be represented by a UML class diagram annotated with OCL invariants.

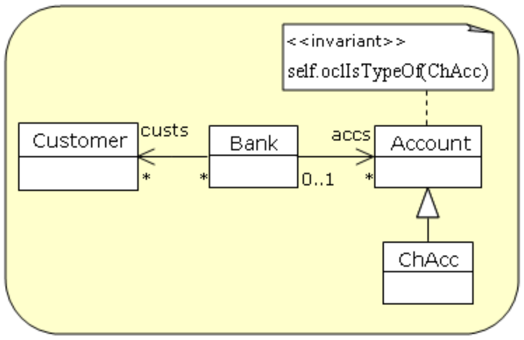

Fig. 1. Banking Analogous UML Class Diagram.

2.2 The Alloy Analyzer

Alloy was simultaneously designed with a fully automatic tool that can simulate models and check properties about them. The tool translates the model to be analyzed into a boolean formula. This formula is transferred to an SAT solver, and the solution is translated back by the Alloy Analyzer into Alloy. The two kinds of analysis consist in binding objects to signatures and relations, searching for a combination of values that make the translated boolean formula true [4].

In particular, simulation generates structures without requiring the user to provide sample inputs or test cases. If the tool finds a binding of objects making the formula true, this binding constitutes a valid snapshot. If we consider, from the previous banking example, the `Bank` and `Account` signatures, along with `accs` relation, a valid snapshot is shown below:

```
Bank = {(B1),(B2),(B3),(B4),(B5)}
Account = {(A1),(A2),(A3),(A4),(A5)}
accs = {(B1,A1),(B1,A2),(B2,A3),(B3,A4),(B5,A5)}
```

In this representation, the Bx and Ax symbols represent Bank and Account objects, respectively. The parentheses are used for tuples (scalar elements are one-element tuples). Notice that every `Account` object is related to at most one

[1] In Alloy, set elements are designed as singleton unary relations.

`Bank` object in `accs`, as modeled in the second formula of the `BankProperties` fact.

On the other hand, assertion checking generates counterexamples - valid snapshots for which an expected property does not hold. The tool searches for a binding of objects which makes true the formula representing the model conjoined with the negated formula of a given logical assertion. A valid snapshot is a counterexample to that assertion. Considering the previous Alloy model, and an assertion stating that each account is associated with one bank (in Alloy: `Account in Bank.accs`), a possible counterexample could be as follows, where the `A3` account is not related to any bank:

```
Bank = {(B1),(B2)}
Account = {(A1),(A2),(A3)}
accs = {(B1,A1),(B2,A2)}
```

The tool does not provide a complete analysis. Instead, it conducts a search within a finite scope chosen by the user, bounding the number of elements in each basic type [4]. For instance, the examples above were analyzed within a scope of five (5). The output is either a snapshot or a message that no snapshot was found in the given scope. When checking a given property, a snapshot is a counterexample and indicates that the asserted formula was not valid. Theoretically, nothing can be inferred when no snapshot is found. However, gradually increasing the scope can give the user a greater confidence during the modeling of critical systems, which helps finding inconsistencies or lack of constraints before implementation. Billions of state combinations for a model's signatures and relations can be covered within a predetermined scope in a matter of seconds, not requiring any input of test cases from the user. This bounded analysis might still constrain models involving numeric types [8].

3 A Semantics for UML Class Diagrams

We offer a semantics for UML class diagrams by translation to correspondent Alloy models. Alloy constructs can represent a number of UML static or dynamic constructs, contributing with a semantics to a subset of UML that may be automatically analyzed. Furthermore, translating a representative subset of OCL expressions to Alloy is straightforward, since they are both defined for expressing constraints in object modeling, based on first-order logic.

In order to define mapping rules between UML and Alloy, we focused on structural properties of class diagrams including OCL invariants, thus exploiting Alloy's expressiveness for modeling complex state properties of a system. For that reason, we did not consider some UML constructs, such as operations (methods) and their effects, besides timing constraints (e.g. {`frozen`}). Although this may seem restrictive, we initially focused on aspects that may be extremely useful in finding problems when modeling complex states. Moreover, Alloy and its supporting tool have also been used for modeling properties over state transitions [6, 9], which shows the language's usefulness in behavioral modeling as well.

We believe that related UML constructs, even other diagrams, can be similarly analyzed by means of analogous mapping.

Regarding OCL, we only consider class invariants, due to the reasons explained above. Since recursive operations have an undefined semantics [10], we do not deal with those in our semantics. Also, numeric types (except integer) are ignored, due to the nature of Alloy's analysis, which is scope-limited. Nevertheless, the latest version of the language introduces an improved support for integer types [8]. Likewise, string type and operations, loop constructs or packaging operations are not considered, since they are implementation-specific (we are primarily concerned with abstract models). The keyword `self` is considered mandatory when expressing context-dependent invariants. We also require role names for each navigable end in binary associations, which simplifies the translation.

Our translation rules are divided into two categories: from UML diagrammatic constructs to Alloy constructs and from OCL invariants to logically-equivalent Alloy formulae. Most UML diagrams do not offer an agreed precise semantics, due to its broad applicability in unlimited modeling contexts [11]. We adapted class diagrams to Alloy semantics (sets of objects with relations as fields), with the purpose of modeling abstract structural aspects of critical systems. Regarding OCL constraints, we based our translation on the OCL specification version 1.5 [10]. Basic set theory and predicate calculus guided the translation rules, neglecting OCL formulae that may present undefined semantics (such as recursion).

Figure 2 depicts a class diagram describing an extended version of the banking system. The invariant over `Customer` states that a customer identifier must be unique, while the invariant over `Account` states that it is an abstract class.

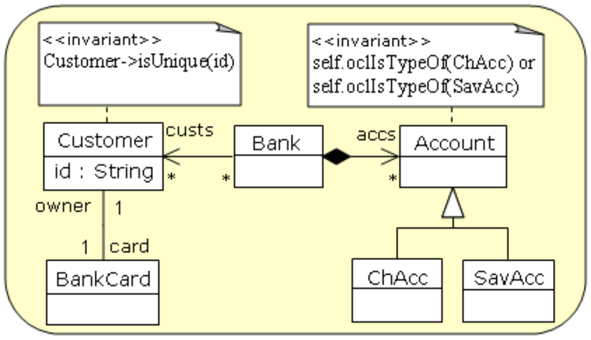

Fig. 2. Extended Class Diagram for the Banking System.

Regarding diagrammatic constructs, classes and interfaces are translated to signatures in Alloy. In addition, binary associations are translated to relations declared with the `set` qualifier. Similarly, attributes also translate to relations. In our example, `Customer` and `BankCard` can be represented in Alloy as follows:

```
sig Customer {
  card: set BankCard,
  id: set String
}
sig BankCard {
  owner: set Customer
}
```

A relation is created for each navigable association end, using the opposite role name. This rule is required due to the limitation in representing navigability constraints with binary relations in Alloy (a binary relation is bidirectional by definition). In case we have two navigable ends for an association, each signature declares a relation for its opposite navigable end, as exemplified by `owner` and `card`. A constraint is added, stating that a relation is the transpose of its opposite counterpart. Furthermore, multiplicity constraints, such as one (1) for `card` and `id`, are expressed as formulae over signatures' relations. The composition between `Bank` and `Accounts` is transformed into a constrained binary relationship. The added multiplicity constraint ('1') on `Bank` represents coincident lifetimes between a bank and its accounts, in addition to forbidden sharing of an account between different banks. These constraints will be all within the `BankProperties` fact. The following Alloy fragment describes this fact.

```
fact BankProperties {
  card = ~owner
  all c:Customer | #(c.card) = 1
  all a:Account | one a.~accs
  ...
}
```

The `#` symbol is the cardinality operator. For example, the second formula in `BankProperties` states that there is exactly one object of `BankCard` mapped by each customer. In addition, generalization is translated to Alloy's `extends`. The subsequent Alloy fragment shows `SavAcc` translated signature:

```
sig SavAcc extends Account {}
```

OCL invariants are translated to equivalent Alloy formulae, being universally quantified on `self`. This limits expression of global properties in OCL, since universally quantified formulae can be true either if the formula is valid or the quantified set is empty. Even though Alloy allows global properties without quantification, we do not intend to fix the problem, as we provide a semantics for OCL.

Next, we show an Alloy fragment that translates the invariants from the `Account` and `Customer` contexts, which will be defined within `BankProperties`. The `oclIsTypeOf` operation is translated to set membership and `isUnique` to an equivalent quantified formula.

```
all self: Account | (self in ChAcc) || (self in SavAcc)
all self: Customer | all disj c,c':Customer | c.id != c'.id
```

The `||` operator corresponds to logical disjunction, while `in` denotes set membership and `!` denotes negation. The `disj` keyword states that the declared variables are distinct. As an example, Table 1 formulates these and aditional mapping rules for OCL expressions, where `X,Y` denote collections, `P,Q` denote logical formulae, `a` and `r` denotes a variable and a attribute, respectively.

Table 1. Examples of Mapping Rules from OCL to Alloy.

OCL	**Alloy**
`X.oclIsTypeOf(Y)`	`X in Y`
`X->includes(Y)`	`Y in X`
`X.allInstances`	`X`
`X.isUnique(r)`	`all disj a,a':X | a.r != a'.r`
`P or Q`	`P || Q`
`P and Q`	`P && Q`
`X->isEmpty()`	`no X`
`X->exists(a|P)`	`some a:X | P`
`X->forAll(a|P)`	`all a:X| P`
`X.size()`	`#X`

Currently, the translation is performed systematically, but manually. Nevertheless, the process was designed to be decidable, allowing automatization. Both languages can be defined by meta-modeling, and transformations can be implemented as a correspondence between meta-model elements, using an approach similar to OMG's Model-Driven Architecture [12]. XML-based representations, such as XMI [13], can certainly help obtaining Alloy paragraphs from UML classes and OCL invariants. Furthermore, OCL constraints can be translated into Alloy formulae as source-to-source transformations, involving a parser for OCL and manipulation of abstract syntax trees. Having an automatic translator from UML to Alloy, constraint solving from the Alloy Analyzer can be provided for a subset of UML class diagrams, as explained in the next section. As the last step of the process, the results yielded by the Analyzer must be transformed back to UML. They could be appropriately represented by object diagrams [1], by integrating instances yielded by the Alloy analyzer with UML CASE tools, such as Rational Rose [14].

4 Class Diagram Analysis

In this section, we present how our approach could be used by developers to improve modeling, adding reliability to the development of critical systems. First, we present, through our banking example, some of the benefits of constraint solving for UML class diagrams. Next, we describe a number of case studies using Alloy and its tool support on modeling of critical systems, which could apply to UML/OCL by our approach.

4.1 Example

Considering the class diagram depicted in Figure 2, a number of unspecified properties could be uncovered by constraint solving in a straightforward way. For instance, if we translate this diagram into an Alloy model with same meaning, by the mapping rules given in Section 3, the simulation of the resulting model may yield the snapshot depicted in Figure 3 (represented as a UML object diagram).

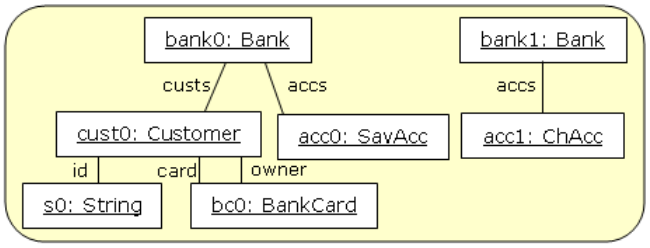

Fig. 3. First Generated Snapshot.

This simulation uses a scope of at most two objects for each entity. The generated snapshot shows that customers and their personal accounts are not related within one bank. Assuming that this is a functional requirement, the model should be modified in order to accommodate the required relationship. The modeler could, for instance, add a directed binary association between these two classes (as depicted in Figure 4), with no multiplicity constraints, then executing a new simulation. A possibly generated snapshot for the new model is showed in Figure 5.

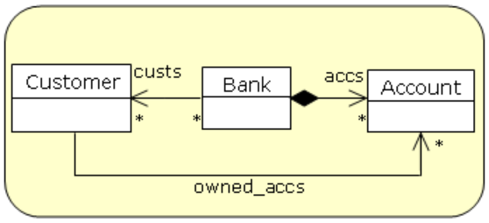

Fig. 4. Modification to the Banking Class Diagram.

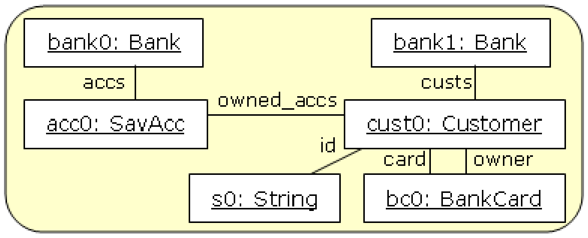

Fig. 5. Second Generated Snapshot.

Although customers and accounts are related, the model allows a snapshot where a customer and its (savings) account are within distinct banks. Assuming that it is a undesirable situation, the modeler can then include the following OCL invariant to the `Bank` context:

```
context Bank inv customersAccountsInBank:
  self.custs.owned_accs->includes(self.accs)
```

which, according to the mapping rules, translates to Alloy:

```
all self:Bank | self.accs in self.custs.owned_accs
```

The translated model possibly yields the snapshot depicted in Figure 6. A number of similar executions, with gradually greater scopes, can increase confidence on the modeled properties.

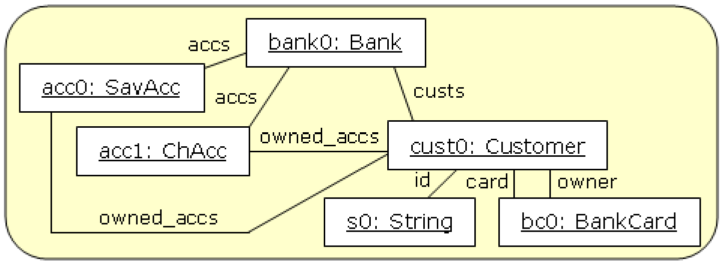

Fig. 6. Third Generated Snapshot.

Additionally, our approach can help identification of over-constrained class diagrams, usually leading to inconsistent models that are not implementable. In the resulting class diagram from the previous example, a functional requirement could be added to the system, allowing customers temporarily without a bank card to be registered into any bank. If we add the following OCL constraint to the `Customer` context:

```
context Customer inv customersNoBankCard:
  Customer.allInstances->exists(c | c.card->isEmpty())
```

No snapshots are found by the Analyzer for the translated Alloy model, even if we increase the scope. This result is due to the newly-added constraint, which contradicts with the multiplicity constraints between `Customer` and `BankCard` objects, stating the existence of an one-to-one correspondence. One of the two constraints must be removed in order to include the desired property to the model.

Finally, one may want to investigate the relationship between `BankCard` and `Account` objects. For a tool implementing our approach, users should be able to enter OCL formulae by means of a dialog box, which may be similar to the USE tool [7]. These formulae would be translated to Alloy assertions, then passed on to the Alloy Analyzer as input. In our example, the following OCL formula could be checked against the diagram:

```
context Customer inv cardsAndAccountsAssertion:
  self.card->notEmpty() implies self.owned_accs->notEmpty()
```

This assertion tests whether every customer that has a card owns at least one account, which could be a new functional requirement. The analysis yields a counterexample, depicted in Figure 7. The snapshot denotes a state with one customer owning a card and no accounts, refuting the assertion. Further changes can fix this requirement, possibly a new constraint relating accounts and customers' bank cards.

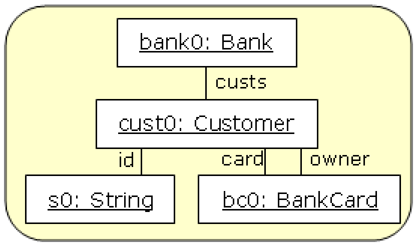

Fig. 7. Generated Counterexample.

Besides showing how our approach can be useful for finding problems in class diagrams, this example shows an additional benefit of this automatic analysis: incremental modeling. A diagram can be written, along with some constraints. Simulation can then give feedback to the modeler regarding the current status of the constraints, indicating the next changes that will accommodate the desired functional requirements. In the same way, assertions offer greater confidence that the model follows those requirements adequately.

4.2 Applications of Alloy and Constraint Solving to Critical Systems

The banking example was used as a simple illustration of our ideas. However, Alloy has been used as a useful approach for finding problems in safety-critical systems. These examples are applicable in the context of this paper, due to the semantics proposed for class diagrams and OCL constraints.

In a recent case study, Alloy was used for specifying a system controlling a radiation therapy machine [6]. This machine produces beams of photons for treating a number of diseases. The system was previously modeled in UML/OCL, and its translation to Alloy uncovered a number of errors. Automatic analysis helped verify whether machine operations were commutative (the order of execution do not affect the final result). It was observed that several pairs of operations did not commute, revealing potential problems during system activity.

Related examples of applying Alloy to critical systems include modeling of access control for information systems [15] and a railway system [9]. In the latter, an Alloy model was built for checking the policy for controlling the motion of trains of the Bay Area Rapid Transit (BART). The analysis was performed in order to check the existence of any condition (state configuration) which could cause train collision.

In addition, an air-traffic control system build by NASA has been modeled using Alloy, which explored new ideas for the design and analysis of such systems [5]. These applications show the potential use of constraint solving and Alloy for understanding critical systems, which can lead to the early discovery of bugs in modeling.

5 Related Work

Our approach allows generation of snapshots and counterexamples to claims over UML class diagrams and OCL invariants, by means of the Alloy Analyzer tool [4]. Related tool support have been developed for similar purposes [16, 17], although allowing syntax and type checking for OCL formulae. In particular, the USE tool [7] also makes OCL machine-analyzable. It contains a useful and easy-to-use model animator, which can uncover bugs by checking snapshots against UML class diagrams constrained with OCL. Our approach differs from USE in the sense that in the latter modelers must provide the verified test cases. Using our approach, snapshots can be automatically generated within a given scope, becoming amenable to automatic simulation. Besides, this analysis allows assertion checking by searching a snapshot that refutes the asserted property (which could be provided as an OCL formula by the user).

Regarding the semantics we provided for UML/OCL, Bordeau and Cheng [18] define a similar approach for a related modeling notation. They automatically map models to algebraic specifications, allowing formal reasoning on the semantics of the translated specification. In contrast, Alloy admits a more direct mapping from UML, since both are similarly suitable to object modeling, as reported by another work [6]. Also, automatic simulation and analysis in Alloy may be more appealing to software architects and designers. In other related approach [19], a systematic approach for translating UML class diagrams with OCL constraints is provided. The translation rules are similar to ours, although they use an additional intermediate language for the translation.

There have also been a number of efforts on proposing formal semantics for UML and related modeling languages, in order to clarify the meaning of its diagrammatic constructs, supporting tool development. For example, related approaches [20, 21] give a formal semantics to a subset of UML class diagrams. We defined a semantics for UML class diagrams and OCL invariants with the specific purpose of leveraging Alloy's automatic analysis to UML.

6 Conclusion

In this paper, we have proposed an approach for automatic analysis of UML class diagrams, according to a semantics based on Alloy, employing the capabilities offered by its analysis tool. This approach provides an option for automation regarding UML/OCL, which can be useful for early identification of problems in critical systems' models, including under- and over-constrained models and undesired properties, which may compromise correctness.

The analysis provided by the Alloy Analyzer is sound but not complete. Since it is based on a user-provided scope, if no snapshot is found, nothing can be inferred (same with counterexamples for assertions). However, small scopes may suffice for improving confidence in modeling and finding relevant problems [22]. Furthermore, the analyzer can generate all possible snapshots for a model within the scope, not requiring any user input. Therefore, many more states can be covered than any testing activity. In this context, Alloy was used as a test case generator for Java programs [23]. We believe that our approach can leverage the benefits of this analysis to UML class diagrams, improving modeling of critical systems using UML. Although the support for numeric types is limited in Alloy, in order to make bounded analysis feasible, a wide range of complex structural properties can still be modeled and analyzed. Such analysis may not even be possible in class diagrams using all features from UML and OCL specifications.

It may be argued that constraint solving might be applied to UML directly, without the need of Alloy as an intermediate language. For example, this can be done through an analysis tool that maps UML class diagrams to a SAT solver, allowing simulation and analysis. However, due to their complex and unresolved semantics, we would still need to map UML and OCL to a formal language, whose boolean formulae are given as input to a SAT solver. For effective analysis, this language should also be sufficiently simple to allow scope-limited analysis over classes and associations. In this context, Alloy is a reasonable choice, since it was developed for constraint solving. Furthermore, we have on-going work on defining an equivalence notion for object models in Alloy, which support the specification of semantics-preserving transformations and refactorings for models [24]. These results can be directly leveraged to UML by our translation to Alloy.

Alloy provides a simple semantics to UML class diagrams and OCL invariants. However, Alloy cannot represent implementation-oriented UML constructs. For instance, attributes are mapped to simple binary relations, disregarding properties such as visibility and default value. Nevertheless, the chosen subset is representative to a number of problems that can be abstractly modelled. Similarly, our approach regards only a subset of UML at first, since Alloy was designed to primarily model structural properties. However, recent work [6, 9] shows use of Alloy for modeling behavioral properties. The extension of our semantics on that topic is considered as future work.

The automation of our approach is an on-going work. The mapping rules between UML/OCL and Alloy can be implemented using well-founded transformation techniques. Analysis is performed on the resulting Alloy model, and the results can be presented as UML object diagrams. This tool support may be integrated to leading UML tools, such as Rational Rose [14], aiming at straightforward application to UML class diagrams with OCL annotations.

7 Acknowledgements

This work benefitted from comments by members of the Software Productivity Group. Also, we give special thanks to Augusto Sampaio, Rodrigo Ramos and Franklin Ramalho for their valuable comments. In addition, we would like to thank all the anonymous referees, whose appropriate suggestions helped improving the paper. This work was partially funded by CAPES and CNPQ, grant 521994/96-9.

References

1. Booch, G., et al.: The Unified Modeling Language User Guide. Object Technology. Addison-Wesley (1999)
2. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Second edn. Addison-Wesley (2003)
3. Jackson, D.: Alloy: A Lightweight Object Modelling Notation. ACM Transactions on Software Engineering and Methodology (TOSEM) **11** (2002) 256–290
4. Jackson, D., Schechter, I., Shlyahter, H.: Alcoa: the Alloy Constraint Analyzer. In: Proceedings of the 22nd International Conference on Software Engineering, ACM Press (2000) 730–733
5. Dennis, G.: TSAFE: Building a Trusted Computing Base for Air Traffic Control Software. Master's thesis, MIT (2003)
6. Dennis, G., Seater, R., Rayside, D., Jackson, D.: Automating Commutativity Analysis at the Design Level. In: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis. (2004) 165–174
7. Richters, M., Gogolla, M.: Validating UML Models and OCL Constraints. In Evans, A., Kent, S., Selic, B., eds.: UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings. Volume 1939 of LNCS., Springer (2000) 265–277
8. Jackson, D.: Alloy 3.0 Reference Manual. Available at http://alloy.mit.edu/beta/-reference-manual.pdf (2004)

9. Jackson, D.: Railway Safety. Available at http://alloy.mit.edu/contributions/-railway.pdf (2002)
10. Object Management Group: OMG Unified Modeling Language Specification Version 1.5 (2003)
11. Gogolla, M., Radfelder, O., Richters, M.: A UML semantics FAQ - the view from bremen. In Kent, S.J.H., Evans, A., Rumpe, B., eds.: Proc. ECOOP'99 Workshop UML Semantics FAQ, University of Brighton (1999)
12. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: the Practice and Promise of The Model Driven Architecture. Addison-Wesley (2003)
13. OMG: XML Metadata Interchange (XMI) Specification (2001) OMG Document formal/02-01-01.
14. Rational Software: Rational Rose XDE Developer (2004) http://www-306.ibm.com/software/awdtools/developer/rosexde/.
15. Zao, J., Wee, H., Chu, J., Jackson, D.: RBAC Schema Verification Using Lightweight Formal Model and Constraint Analysis. Submitted to SACMAT 2003 (2002)
16. Tigris.org: ArgoUML (2004) http://argouml.tigris.org/.
17. Klasse Objecten: Octopus: OCL Tool for Precise Uml Specifications (2004) http://-www.klasse.nl/ocl/octopus-intro.html.
18. Robert H. Bourdeau and Betty H. C. Cheng: A Formal Semantics for Object Model Diagrams. IEEE Transactions on Software Engineering **21** (1995) 799–821
19. Lano, K., Clark, D., Androutsopoulos, K.: UML to B: Formal verification of object-oriented models. In Boiten, E.A., Derrick, J., Smith, G., eds.: Proceedings, International Conference on Integrated Formal Methods, Canterbury, UK. Volume 2999 of LNCS., Springer (2004) 187–206
20. Clark, T., Evans, A., Kent, S.: The Metamodelling Language Calculus: Foundation Semantics for UML. In Hussmann, H., ed.: Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2-6, 2001. Volume 2029 of LNCS., Springer (2001) 17–31
21. Evans, A., France, R.B., Lano, K., Rumpe, B.: The UML As a Formal Modeling Notation. In Bézivin, J., Muller, P.A., eds.: The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998. Volume 1618 of LNCS., Springer (1999) 336–348
22. Khurshid, S., Marinov, D., Jackson, D.: An Analyzable Annotation Language. In: Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ACM Press (2002) 231–245
23. Marinov, D., Khurshid, S.: TestEra: A Novel Framework for Automated Testing of Java Programs. In: Proceedings of the 16th IEEE International Conference on Automated Software Engineering, IEEE Computer Society (2001) 22
24. Gheyi, R., Borba, P.: Refactoring Alloy Specifications. In Cavalcanti, A., Machado, P., eds.: Electronic Notes in Theoretical Computer Science, Proceedings of the Brazilian Workshop on Formal Methods. Volume 95., Elsevier (2004) 227–243