Semantic Comparisons of Alloy Models

Jan Oliver Ringert
University of Leicester
UK

Syed Waqee Wali
University of Leicester
UK

ABSTRACT

Alloy is a textual modeling language for structures and behaviors of software designs. The Alloy Analyzer provides various analyses making it a popular light-weight formal methods tool. While Alloy models can be analyzed, explored, and tested, there is little support for comparing different versions of Alloy models. We believe that these comparisons are crucial when trying to refactor, refine, or extend models. In this work we present an approach for the semantic comparisons of Alloy models. Our pair-wise comparisons include semantic model differencing and the checking of refactoring, refinement, and extension. We enable semantic comparisons of Alloy models by a translation of two versions into a single model that is able to simulate instances of either one or of the versions. Semantic differencing and instance computation require only a single analysis of the combined model in the Alloy Analyzer. We implemented our work and evaluated it using 654 Alloy models from different sources including version histories. Our evaluation examines the cost of semantic comparisons using the Alloy Analyzer in terms of running times and variable numbers over individual models.

CCS CONCEPTS

• **Software and its engineering → Specification languages**; **Semantics**.

KEYWORDS

Alloy, Specification, Differencing, Semantics, Evolution

ACM Reference Format:
Jan Oliver Ringert and Syed Waqee Wali. 2020. Semantic Comparisons of Alloy Models. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3365438.3410955

1 INTRODUCTION

Alloy [14–16] is a textual modeling language based on relational first-order logic. Alloy models declaratively express structures and behaviors of software designs. The Alloy Analyzer[2] provides various analyses for finding instances or counterexamples for Alloy models. Analyses are possible in a bounded scope and fully automated by translations to SAT solvers making Alloy a popular light-weight formal method [16].

MODELS '20, October 18–23, 2020, Virtual Event, Canada

ACM ISBN 978-1-4503-7019-6/20/10…$15.00
https://doi.org/10.1145/3365438.3410955

Applications of Alloy cover various case studies. Popular examples include Zave's [35] analysis of the Chord protocol, security related analyses of single sign-on protocols [1], and the Mondex electronic purse [28]. Other works leverage Alloy as a back-end solver, e.g., for formalizing class diagrams [3, 7, 22] or software architecture models [5, 18] and many more[1]. However, Alloy has also received much attention from researchers beyond creating Alloy models. The Alloy language has been extended in various ways. As an example, Alloy* [24] extends Alloy with higher-order quantification and Electrum [6] extends Alloy with temporal logic.

Importantly, researchers have addressed issues of making Alloy itself more convenient to use. As an example, Alloy has been extended with support for partial instance declaration and validation [25]. As another example, AUnit [30] is part of a series of works on testing tools for Alloy models [30–32]. Others worked on finding minimal instances [27] and on analyzing provenance of instances [26].

While individual Alloy models can be analyzed, explored, and tested, there is little support for comparing different versions of Alloy models. Due to the textual nature of Alloy models, they can be compared using traditional differencing tools [29]. However, applied to specifications with non-trivial constraints, traditional syntax-based differencing has its limits. On the one hand, small syntactical changes to an Alloy model may have a large impact on its semantics in terms of possible instantiations. On the other hand, large syntactical changes where parts of a model are rewritten might simply be refactorings to improve readability or analyses times. In these cases an engineer might wish for semantic comparisons of Alloy models, e.g., semantic differencing [19–21, 23] to inspect instances that were added to the semantics or a refactoring check to confirm that a model's semantics were preserved.

We present a method for semantic comparisons of Alloy models. This includes semantic differencing, refinement, extension, and equivalence checks. We leverage Alloy itself for the analyses and computation of instances. Specifically, we present a translation of two (versions of) Alloy models into one. We show that in most cases the expressiveness of the Alloy language is enough for encoding semantic comparisons of Alloy models. We identify limitations in Sect. 5.2, e.g., Alloy's partial support for higher-order quantification.

We have implemented our work and evaluate it using 654 Alloy models from different sources including version histories. Our evaluation examines the cost of semantic comparisons using the Alloy Analyzer in terms of running times and variable numbers of the resulting SAT problems.

The remainder of this paper is organized as follows. Sect. 2 introduces an example of Alloy models and motivates their semantic comparison. Sect. 3 presents preliminaries. Sect. 4 details our translation, Sect. 5 shows how analyses are enabled and discusses current

[1] See the incomplete list at https://alloytools.org/citations/language-translations.html

```
1  sig Endpoint {}
2  abstract sig Request {
3    to: set Endpoint
4  }
5  sig LoginRequest extends Request {
6    from: one Endpoint
7  }
```

Listing 1: Initial model $v1$ of endpoints, requests and login requests.

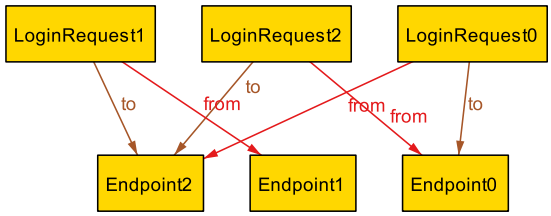

Figure 1: Instance $i1$ of model $v1$

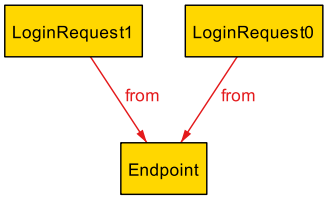

Figure 2: Instance $i2$ of model $v1$ showing some undesired properties, e.g., missing targets of requests

limitations. Sect. 6 presents an evaluation, Sect. 7 discusses related work and finally, Sect. 8 concludes.

2 EXAMPLE

We present an example inspired by the web security model from [1, 16]. The example Alloy model $v1$ shown in Lst. 1 models login requests. Concretely, the model contains three signatures for network endpoints, requests, and login requests. Signature `LoginRequest` extends signature `Request` and inherits field `to`. Login requests can go from one endpoint (Lst. 1, l. 6) to multiple endpoints (l. 3).

The semantics of the Alloy model $v1$ contains instances that engineers can explore. The instance shown in Fig. 1 shows three login requests all from different endpoints. Each request goes to one endpoint. The engineers further explore instances of model $v1$ and find the one shown in Fig. 2 of two login requests from the same endpoint without any target. This instance does not represent a valid scenario and the engineering team decides to update the model. Every endpoint should make at most one login request to exactly one endpoint.

In the next meeting two engineers present revised versions of model $v1$: model $v2$ shown in Lst. 2 and model $v3$ shown in Lst. 3. Syntactically $v2$ and $v3$ look quite different from model $v1$ and even more so from one another. As an example, model $v2$ uses a signature fact to constrain the number of targets of each login request (Lst. 2, l. 8) and model $v2$ is missing signature `Request` from model $v1$.

```
1  sig Endpoint {}
2  abstract sig Request {
3    to: set Endpoint
4  }
5  sig LoginRequest extends Request{
6    from: one Endpoint
7  }{
8    #to = 1  // one target for each request
9  }
10 fact { // from is injective
11   all r1, r2 : LoginRequest |
12     r1.from = r2.from implies r1 = r2
13 }
```

Listing 2: Model $v2$ with a signature fact on the size of relation to and a fact about sources of login requests

```
1  sig Endpoint {}
2  sig LoginRequest {
3    to: one Endpoint,
4    from: one Endpoint
5  }
6  fact {
7    from in LoginRequest lone -> one Endpoint
8  }
```

Listing 3: Model $v3$ removed the abstract signature `Request` from $v1$ and restricts multiplicities

The engineers use our tool to establish that indeed model $v2$ is a **refinement** of $v1$, i.e., $v2$ only removes instances from the semantics of $v1$ and does not add any (unwanted) instances. To gain confidence in the correctness of the instances removed, the engineers explore the **semantic difference** $diff(v2, v1)$ of instances of $v1$ that are not instances of $v2$. Indeed all instances in $diff(v2, v1)$, e.g., $i2$ shown in Fig. 2, are undesired instances. The team also explores **common instances** of $v1$ and $v2$ and finds that $i1$ from Fig. 1 is preserved.

Finally, the engineers use our tool to confirm, to their surprise, that the models $v2$ and $v3$ have **equivalent semantics**. All instances of model $v1$ are also instances of model $v2$ and vice versa. They decide to keep model $v2$ because they like the obvious pattern in Lst. 2, ll. 11-12 known from injective functions and they want to keep the flexibility of the abstract signature `Request`. In addition, model $v2$ generates smaller SAT problems than model $v3$, which usually is beneficial for analysis times using the Alloy Analyzer.

3 PRELIMINARIES

3.1 Models and Semantics

Harel and Rumpe [13] define a modeling language $\mathcal{L} = \langle \mathcal{M}, \mathcal{S}, sem \rangle$ as a tuple where $\mathcal{M}$ is the set of well-formed models according to some syntax definition, $\mathcal{S}$ is a semantic domain, and $sem : \mathcal{M} \rightarrow \wp(\mathcal{S})$ defines the meaning of a model $m \in \mathcal{M}$ by mapping the model to a set of elements of $\mathcal{S}$ (see [13]).

Maoz et al. [19, 20] have introduced semantic differencing where models are compared in terms of their semantics rather than syntax. Their differencing operator for two models $v1$ and $v2$ is defined as $diff(v1, v2) = sem(v2) \setminus sem(v1)$.

3.2 Alloy

Alloy [14–16] is a textual modeling language based on relational first-order logic. Alloy's syntax consists of signatures, fields, facts, functions, and predicates. Expressions offer Boolean logic, limited support for arithmetic, and first order quantification. The semantic domain of Alloy models are its set of instances. These are models of bounded first order relational logic. Each signature denotes a set of atoms, which are the basic entities in Alloy. The semantics of fields are relations between two or more signatures. Instances of fields are sets of tuples of atoms. Facts are statements that define constraints on the elements of the model. Predicates and functions are parametrized constraints. They can be included in other predicates, functions, or facts.

We present a simplified abstract syntax of an Alloy model in Def. 1. The definition is based on the code of Alloy Tools 5.10 available from [2]. Here, the set of signatures $m.sigs$ and facts $m.facts$ refer to all user-defined signatures and facts, including those defined in imported models[2].

Definition 1 (Alloy abstract syntax, simplified from [2, 15]). The abstract syntax of an Alloy model m consists of

- $m.sigs$ a set of signatures, where each signature s has
  - $s.name$ a unique, qualified signature name
  - $s.mult$ a multiplicity **lone**, **some**, or **one**
  - $s.parent/s$ a declaration of parent signatures[3]
  - $s.fields$ a set of fields, where each field has
    - $f.name$ a name unique for the signature
    - $f.mult$ a multiplicity **lone**, **some**, **one**, or **set**
    - $f.expr$ an expression defining the field's values
  - $s.facts$ a set of facts, possibly referencing fields of s
- $m.facts$ a conjunction of all facts of the model
- $m.commands$ a set of commands

We denote the semantics of Alloy model m by $[\![]\!]_m$. Given a constraint c, we denote by $[\![\mathsf{c}]\!]_m$ the elements of the semantics of model m that satisfy constraint c.

Alloy models can be analyzed using the Alloy Analyzer, a fully automated constraint solver. Each analysis starts from a run or check command cmd with a constraint $cmd.formula$. Intuitively, the Alloy Analyzer computes instances that are elements of $[\![cmd.formula]\!]_m$. The Alloy instances are relations of atoms. Every signature of the model is represented by a unary relation (set) of atoms and every field is represented by an n-ary relation over the same atoms. The automated analysis is done by a translation of the model into a Boolean expression, which is analyzed by SAT solvers. The analysis is based on an exhaustive search for instances of the model, bounded by a user-specified scope, which limits the number of atoms for each signature in an instance of the system that the solver analyzes.

4 TRANSLATION OF MODEL PAIRS

To realize the semantic comparison of Alloy models, we translate two models $v1$ and $v2$ into a single model $T(v1, v2)$ where instances of $v1$ or $v2$ are also instances of $T(v1, v2)$, formally $[\![.]\!]_{v1} \cup [\![.]\!]_{v2} \subseteq [\![.]\!]_{T(v1,v2)}$. Furthermore, by generating model-specific constraints cV1 and cV2 that express restrictions in $T(v1, v2)$ implied by $v1$ and $v2$, we are able to restrict instances of $T(v1, v2)$ to those of $v1$ and $v2$, formally $[\![.]\!]_{v1} \equiv [\![\mathsf{cV1}]\!]_{T(v1,v2)}$ and $[\![.]\!]_{v2} \equiv [\![\mathsf{cV2}]\!]_{T(v1,v2)}$. Apart from generating constraints cV1 and cV2, a main challenge of our approach is to structurally enable a merge of models. The models $v1$ and $v2$ may contain signatures with the same names but different fields or different cardinalities.

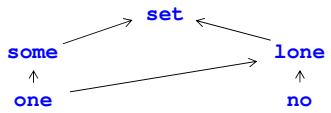

Figure 3: Partial order of Alloy multiplicities to computer least upper bounds

A naive approach would be to create all signatures with **set** cardinality and to add all fields from $v1$ and $v2$ with all types changed to **univ**, Alloy's parent type of all types. The constraints cV1 and cV2 would then restrict these most permissive structures to the semantics of each model. However, this approach is very likely to be infeasible as the underlying relational model would almost have no structural restrictions — leading to very large SAT problems.

The main idea of our translation is to create signatures and fields that represent small upper bounds for signatures and fields of $v1$ and $v2$. We now go over the main language features of Alloy models and describe how they are translated by our translation $T(v1, v2)$.

4.1 Signatures and Fields

Given two models $v1$ and $v2$, Alg. 1 creates the set of signatures of model $T(v1, v2)$ and updates constraints cV1 and cV2 to restrictions of the respective source models. The algorithm first handles common signatures and then signatures unique to either $v1$ or $v2$.

Common signatures $s1$ and $s2$ are merged into a new signature s. The multiplicity of s is the **leastUpperBound** of the multiplicities of $s1$ and $s2$ (Alg. 1, l. 4). We compute the least upper bound based on the partial order of Alloy multiplicities shown in Fig. 3. This step makes sure that both multiplicities can be represented in the semantics of $T(v1, v2)$. As an example, the least upper bound of multiplicities **one** (exactly one) and **lone** (zero or one) is **lone**, and the least upper bound of **some** (at least one) and **lone** is **set**. Our algorithm adds the original multiplicity restrictions to cV1 and cV2 in ll. 5, 12.

The structure for unique signatures is similar (Alg. 1, ll. 9-16). Atoms of unique signatures are prevented from appearing in the semantics in the second model (Alg. 1, ll. 11, 12).

Finally, signature facts from $v1$ or $v2$ are not added to the new signatures (Alg. 1, ll. 7,14) as they might differ in each model. We add these to the constraints cV1 and cV2 in Alg. 1, ll. 17-22. Note that the quantified variable `this` is used inside the facts of signatures to relate to each atom [15].

The translation **transExpr** used in Alg. 1 is necessary as expressions in Alloy's abstract syntax reference signatures, predicates, etc. from models $v1$ and $v2$ that no longer exist in $T(v1, v2)$. We describe **transExpr** in Sect. 4.3. The field $s.parent$ is left empty as

[2]The corresponding Java methods in class `CompModule` of Alloy are `getAllReachableUserDefinedSigs()` and `getAllReachableFacts()`
[3]Alloy's type signatures have a single parent $s.parent$ and Alloy's subset signatures have a set of parents $s.parents$

Algorithm 1 Signature merge algorithm for models $v1$ and $v2$

1: $sigs = \emptyset$
2: **for** $s1 \in v1.sigs$, $s2 \in v2.sigs$ where $s1.name = s2.name$ **do**
3: create new $s \in sigs$ with $s.name = s1.name$
4: $s.mult$ = **leastUpperBound**($s1.mult$, $s2.mult$)
5: cV1 = (cV1 **and** $s1.mult$ s) and cV2 = (cV2 **and** $s2.mult$ s)
6: $s.fields$ = **mergeFields**($s1.fields$, $s2.fields$)
7: $s.facts = s.parent = \emptyset$
8: **end for**
9: **for** $sig \in v1.sigs \cup v2.sigs$ with $sig.name$ unique in $v1$ **do**
10: create new $s \in sigs$ with $s.name = sig.name$
11: $s.mult$ = **leastUpperBound**($sig.mult$, **no**)
12: cV1 = (cV1 **and** $sig.mult$ s) and cV2 = (cV2 **and no** s)
13: $s.fields$ = **mergeFields**($sig.fields$, $\emptyset$)
14: $s.facts = s.parent = \emptyset$
15: **end for**
16: {same for signatures unique in $v2$}
17: **for** $sig \in v1.sigs$ with corresponding $s \in sigs$ **do**
18: **for** $fact \in sig.facts$ **do**
19: cV1 = cV1 **and** (**all this** : s | **transExpr**($fact$))
20: **end for**
21: **end for**
22: {same for signature facts from $v2$}
23: **return** $sigs$

inheritance structures might be different in $v1$ and $v2$. We show how we enable inheritance in Sect. 4.2.

We translate fields (**mergeFields**($s1$, $s2$) in Alg. 1) according to Alg. 2. The algorithm has a similar structure to Alg. 1 and first handles common fields and then fields unique in $s1$ or $s2$.

Again, field multiplicities of new fields f are calculated using **leastUpperBound**($f1.mult$,$f2.mult$). The type of a field f is the union of the types of $f1$ and $f2$. Types are implicitly defined by expressions $f1.expr$ and $f2.expr$. These expressions may be quite powerful, e.g., we could define field `from` in Lst. 1 as `from: one (Endpoint - to)` to exclude all destinations as the sender. Expressions in field declarations may reference fields of the same or an inherited signature. Supporting these expressions again requires the quantification over atoms and field restrictions inside cV1 and cV2 shown in Alg. 2, ll. 7-8, 14. Note that the constraint inside the quantification restricts both the multiplicity and the type of the field, e.g., **this.** f **in** $f1.mult$ $e1$ (Alg. 2, l. 7) restricts the new field f to the multiplicity of the original field $f1.mult$ and to the original but translated type expression $e1$.

The type of a field f may have any arity, e.g., it can hold tuples of atoms. For arities greater than one, fields do not have multiplicities $f.mult$, but the multiplicities of the relation are declared inside $f.expr$ with product operators (arrows) — as in our example in Lst. 3 `LoginRequest lone -> one Endpoint`. Alloy's **union** operator requires identical multiplicities of relations. Thus, we replace arrows with multiplicities in $e1$ and $e2$ by the generic product arrow in Alg. 2, l. 6.

Finally, fields that appear only in one signature are modified to disappear in the semantics of the other model in Alg. 2, ll. 10-17.

Algorithm 2 Merging of fields of common signatures $s1$ and $s2$

1: $fields = \emptyset$
2: **for** $f1 \in s1.fields$, $f2 \in s2.fields$ with same name **do**
3: create new $f \in fields$ with $f.name = f1.name$
4: $f.mult$ = **leastUpperBound**($f1.mult$, $f2.mult$)
5: $e1$ = **transExpr**($f1.expr$) and $e2$ = **transExpr**($f2.expr$)
6: $f.expr$ = **union**($e1$, $e2$) {for higher arities replace arrows}
7: cV1 = cV1 **and** (**all this** : **transExpr**($s1$) | **this.** f **in** $f1.mult$ $e1$)
8: cV2 = cV2 **and** (**all this** : **transExpr**($s2$) | **this.** f **in** $f2.mult$ $e2$)
9: **end for**
10: **for** $f1 \in s1.fields$ with unique name **do**
11: create new $f \in fields$ with $f.name = f1.name$
12: $f.mult$ = **leastUpperBound**($f1.mult$, **no**)
13: $f.expr$ = **transExpr**($f1.expr$)
14: cV1 = cV1 **and** (**all this** : **transExpr**($s1$) | **this.** f **in** $f1.mult$ **transExpr**($f1.expr$))
15: cV2 = **no** f
16: **end for**
17: {same for fields unique in $s2$}
18: **return** $fields$

4.2 Inheritance and Subset

Alloy provides two kinds of signatures: type signatures and subset signatures. The instances of type signatures are sets of atoms that may be shared by other type signatures in an inheritance hierarchy. Inheritance is declared by keyword **extends** and a single, direct parent signature. Type signatures inherit *downwards* all fields and facts from their parent. All signatures in our examples Lst. 1-3 are type signatures. On the contrary, subset signatures are declared using keyword **in** instead of **extends** and can list multiple parents. Subset signatures may add fields *upwards* to any subset of atoms of their parents. To make things more interesting, fields and facts added *upwards* by subset signatures are inherited *downwards* by type signatures.

Intuitively, to support both in our translation, we collect all inherited or added fields for signatures in each model in Alg. 3. The algorithm also creates mappings that translate between signatures in $v1$ and $v2$ and a flattened inheritance in $T(v1, v2)$.

We cannot rebuild the inheritance hierarchy of models $v1$ and $v2$ in model $T(v1, v2)$ as Alloy only supports single inheritance and forbids inheritance cycles. Hence, we flatten the hierarchy by copying fields to all direct and indirect subsignatures. This set of fields is computed in Alg. 3, l. 3 where $parent^*$ denotes the transitive closure of the parent signature relation. Furthermore, whenever a signature is referenced, we replace that reference by the union of subsignatures (mapping computed in Alg. 3, l. 12), where $s', sub' \in sigs$ are the signatures corresponding to $s, sub \in v1.sigs$ as created in Alg. 1. Whenever a field is referenced, we replace that reference by the union of all its copies in subsignatures (omitted in Alg. 3 but similar to signatures).

Similarly, subset signatures might have different parents or fields in $v1$ and $v2$ and cannot directly be added to $T(v1, v2)$. However, in $T(v1, v2)$ we need to keep track of the atoms of parents that are

contained in subset signatures, in order to add additional fields or constraints. This is due to the semantics of subset signatures where any instance of a parent may be in the subset signature, e.g., some instances of parent signatures have additional fields provided by the subset signature and other instances do not. Again, we copy fields of subset signatures to all of their parents' type signatures (computed in Alg. 3, ll. 4-6). Note that l. 4 implements an *upwards* propagation of fields, i.e., $s \in sub.parents^*$ inherits fields from subset signatures *sub*, and an *upwards-downwards* propagation of fields, i.e., s inherits fields from its type signature parents $p \in s.parent^*$ that these inherited from subset signatures *sub*. Finally, Alg. 3 extends mapping $sigExV1$ for subset signatures in ll. 14-16. A subset signature $s \in v1.sigs$ is simply mapped to its corresponding subset signature $s' \in sigs$.

Algorithm 3 Computing inherited fields and sig references

```
1:  create map finV1 from name to fields
2:  for type sig s ∈ v1.sigs do
3:      fields = s.fields ∪ s.parent*.fields
4:      for subset sig sub ∈ v1.sigs s.t. either s ∈ sub.parents* or
                                  ∃p ∈ s.parent* s.t. p ∈ sub.parents* do
5:          fields = fields ∪ sub.fields
6:      end for
7:      finV1(s.name) = fields
8:  end for
9:  {same for finV2 for fields in v2}
10: create map sigExV1 from name to signatures
11: for type sig s ∈ v1.sigs with corr. s' ∈ sigs do
12:     sigExV1(s.name) = s' ∪ {sub' ∈ sigs | s ∈ sub.parent*}
13: end for
14: for subset sig s ∈ v1.sigs with corr. s' ∈ sigs do
15:     sigExV1(s.name) = s'
16: end for
17: {same for sigExV2 for signatures in v2}
18: return maps finV1, finV2, sigExV1, sigExV2
```

The merge of type signatures remains as in Alg. 1. However, for subset signatures, the merge is slightly modified: First, fields in Alg. 1, l. 6, 13 are not added for subset signatures (we copied all fields to type signatures in Alg. 2, ll. 4-6). Second, we require Alloy to keep track of atoms that are members of subset signatures. Thus, for signature s in l. 7 we set $s.parents$ to the union (least upper bound) of parent type signatures of $s1$ and $s2$ (in l. 14 we set $s.parents$ to the parent type signatures of $s1$). Finally, we extend the constraints `cV1` and `cV2` to ensure containment of child subset signatures in the union of their parents in either model.

Based on the mappings computed by Alg. 3, we update Alg. 2 of field creation to create the additional, inherited fields. Specifically, we replace $s1.fields$ by $finV1(s1.name)$ and $s2.fields$ by $finV2(s2.name)$. For fields added by subset signatures, we allow multiplicity **no** in Alg. 2, l. 4 (as already done in l. 12). To ensure that fields added by subset signatures are only populated if an atom is contained in the subset signature we extend the conjunct added to `cV1` and `cV2` in Alg. 2, l. 7-8, 14. Specifically, for field f contributed by subset signature *sub*, we extend the body of the universal qualification to **this in** *sub* **implies** [..] **else no this**.f, where [..] is the original expression from Alg. 2, l. 7-8, 14.

Finally, abstract signatures s from $v1$ with at least one subsignature in $v1$ are omitted from $T(v1, v2)$ (similarly for $v2$).

We use the mappings $sigExV1$ and $sigExV2$ in Sect. 4.3.

4.3 Expressions: Facts, Functions, ...

The abstract syntax of Alloy models is quite tangled with references to syntax elements, e.g., the abstract syntax of expression `#to = 1` in Lst. 2 has a reference to a quantified variable **this** of type `LoginRequest` and a reference to field `to` of signature `Request` that references signature `Endpoint`. All these references that are valid in the abstract syntax of $v2$ are invalid in the abstract syntax of $T(v1, v2)$ and have to be replaced.

Our translation applies **transExpr**() shown in Alg. 4 to every expression. The input is an expression from either $v1$ or $v2$ and the output is an equivalent expression valid in $T(v1, v2)$. We show the translation of expressions from $v1$ in Alg. 4.

Algorithm 4 Translation **transExpr**(e) of expression e from $v1$ to $T(v1, v2)$ using mappings from Alg. 3

```
1:  switch(kind of e)
2:      case(constant):
3:          return e
4:      case(unary expression):
5:          return e.op transExpr(e.sub)
6:      case(signature):
7:          return sigExV1(e.name)
8:      case(fun/pred call):
9:          args = ∅
10:         for arg in e.args
11:             add transExpr(arg) to args
12:         end for
13:         return create call of transExpr(e.fun) with args
14:     ...
```

The first two cases of constants and unary expressions are straight forward. Note that the second case relies on a typical recursive call of Alg. 4 to itself on subexpressions. The third case demonstrates how references to signatures are replaced by their corresponding expressions computed in Alg. 3 to support inheritance and subset signatures. The last example in Alg. 4 is the translation of calls of predicates and functions. Again, all subexpressions, i.e., arguments and the function itself, are translated individually by further invocations of Alg. 4.

4.4 `run` and `check` Commands

Alloy supports two kinds of commands: **run** commands for finding instances of models and **check** commands for finding counterexamples to assertions on the model. Internally, run commands consist of a predicate and optional scope declarations. In Alloy's abstract syntax (see Def. 1 and [15, App. B]), run commands consist of all constraints of the model conjoined with the run predicate of the command (including necessary quantification over the parameters of the run command).

Supporting run commands from $v1$ and $v2$ with predicates p_1 and p_2 is straight forward: we replace cV1 by cV1 **and** p_1 and we replace cV2 by cV2 **and** p_2.

Comparing two check commands with assertions a_1 and a_2 is possible in the same way (assertions are also predicates in Alloy's abstract syntax). However, note that the evaluations of the semantic difference then are counterexamples that violate assertion a_2 and satisfy assertion a_1. This is semantically correct, but might not be very intuitive.

User-Defined Scopes. As part of any command, a user may declare scopes for signatures. Again, our translation cannot use these scopes as is for $T(v1, v2)$, as they might be conflicting for common signatures. Also note that there are many rules for implicit scopes, inheritance, and overriding of scopes.

To support scopes in commands, we let Alloy compute detailed per-signature scopes on the original models. For every signature $s1 \in v1.sigs$ with scope $scope(s1)$[4] we extend cV1 with conjunct

$$\#(\textbf{transExpr}(s1)) \ \texttt{<=} \ scope(s1)$$

In case a scope is exact, the operator <= is replaced by =. Note the use of **transExpr**() from Alg. 4 to take care of inheritance by replacing parent signatures by the union of their subsignatures.

The scope of each signature added to the command for model $T(v1, v2)$ is the maximum scope from models $v1$ and $v2$.

4.5 Complete Translation

The complete translation of models $v1$ and $v2$ to $T(v1, v2)$ is as follows:

- $T(v1, v2).sigs$ is computed by Alg. 1 to Alg. 4
- $T(v1, v2).facts = \emptyset$ as all constraints are in cV1 and cV2
- $T(v1, v2).commands$, if needed, as in Sect. 4.4, but depending on the chosen analysis, see Sect. 5.1

An example translation output $T(v1, v2)$ translated from models $v1$ in Lst. 1 and $v2$ in Lst. 2 (all without commands) is shown in Lst. 4 where cV1 and cV2 are visualized as predicates.

We observe the following upper bounds for the size of the output of our translation. The number of signatures $|T(v1, v2).sigs|$ is at most $|v1.sigs| + |v2.sigs|$ (they are equal if all signatures are unique), see Alg. 1. The number of all fields $|T(v1, v2).sigs.fields|$ is at most $|v1.sigs.fields| \cdot |v1.sigs| + |v2.sigs.fields| \cdot |v2.sigs|$ due to the blow-up from copying fields in Alg. 3. Alg. 1-Alg. 3 add $O(|v1.sigs.fields| \cdot |v1.sigs| + |v2.sigs.fields| \cdot |v2.sigs|)$ additional constraints (again due to the blow-up from copying fields in Alg. 3). Finally, Alg. 4 might increase the size of expressions by factor $max(|v1.sigs|, |v2.sigs|)$ when replacing the reference to a signature with the union of its subsignatures computed in Alg. 3.

5 ANALYSES AND LIMITATIONS

5.1 Example Semantic Analyses

Our translation presented in Sect. 4 enables different semantic analyses of pairs of Alloy models. By construction of our translation $T(v1, v2)$, an Alloy instance that satisfies predicate cV1 (cV2) is in the semantics of model *v1* (*v2*), formally, $[\![.]\!]_{v1} \equiv [\![\text{cV1}]\!]_{T(v1,v2)}$ (and respectively $[\![.]\!]_{v2} \equiv [\![\text{cV2}]\!]_{T(v1,v2)}$). We can thus compute the following example semantic analyses of Alloy models.

[4]We use class ScopeComputer of Alloy for this task

```
sig Endpoint {}
sig Request {
  to: set Endpoint
}
sig LoginRequest {
  to: set Endpoint,
  from: one Endpoint
}
pred cV1 {
  no Request
  #(Request + LoginRequest) <= ... //Sect.4.4
}
pred cV2 {
  all this : LoginRequest | #this.to = 1
  all r1, r2 : LoginRequest |
    r1.from = r2.from implies r1 = r2
  no Request
  #(Request + LoginRequest) <= ... //Sect.4.4
}
```

Listing 4: Model $T(v1, v2)$ translated from models $v1$ in Lst. 1 and $v2$ in Lst. 2

Note that the semantics operator $[\![\,]\!]$, when running analyses in Alloy, is scope dependent. Any semantic analysis in Alloy requires a command and we must restrict scopes as shown in Sect. 4.4.

Semantic Difference. To compute an instance from the semantic difference of $diff(v1, v2)$ we evaluate $[\![\text{cV2 **and not** cV1}]\!]_{T(v1,v2)}$.

Refinement. To check whether $v2$ refines $v1$, i.e., to check whether $[\![.]\!]_{v2} \subseteq [\![.]\!]_{v1}$, we check that $[\![\text{cV2 **and not** cV1}]\!]_{T(v1,v2)}$ has no instance. Note that this check is the same as the computation of the semantic difference.

Extension. To check whether $v2$ extends $v1$, i.e., to check whether $[\![.]\!]_{v1} \subseteq [\![.]\!]_{v2}$, we check that $[\![\text{cV1 **and not** cV2}]\!]_{T(v1,v2)}$ has no instance. Note that extension is the dual to refinement.

Equivalence. To check whether $v2$ equals $v2$, i.e., to check whether $[\![.]\!]_{v1} \equiv [\![.]\!]_{v2}$, we check that $[\![\textbf{not}(\text{cV1 **iff** cV2})]\!]_{T(v1,v2)}$ has no instance. A similar pattern was used in an example in [34].

Scope. Recall that Alloy operates in a bounded scope and that all analyses are only valid in that scope, i.e., two models could be equivalent in a smaller scope, when they are not for a larger scope. A trivial case of equivalence is where the scope is too low to satisfy neither cV1 nor cV2.

5.2 Current Limitations

The translation presented in Sect. 4 has limitations due to syntactic restrictions of Alloy, due to the nature of some of the enabled analyses, and due to implementation details of internal predicates inside Alloy.

Field arities. First, merging fields of the same signature with the same name but different arities, e.g., unary and binary, is not supported. This is due to the restriction that a field must have an n-ary relation type. One could imagine padding the relations

```
sig A {}
one sig Ord{
   First: set A,
   Next: A -> A
}
run {pred/totalOrder[A,Ord.First,Ord.Next] and
    !pred/totalOrder[A,Ord.First,Ord.Next]}
   for exactly 3 A
```

Listing 5: An Alloy model demonstrating issues with the discouraged use of built-in predicate `totalOrder`. Against our intuition this model has instances

with the lower arity to the arity of the second one. However, this would require a different interpretation of instances produced by the analyses that no longer correspond one-to-one to instances of the original models.

Signature kind clash. Second, Alloy provides type and subset signatures. Our translation cannot merge signatures of different kinds with identical names. Note that this only applies to the signatures itself, e.g., we do support merging fields of type signatures with fields of subset signatures.

Higher-order quantification. Third, Alloy has limited support for higher-order quantification through Skolemization [15]. As an example, the quantification in `some x : set Person | Person.children = x` is handled well by Alloy although it is higher-order as `x` is a set. However, the negation of this higher-order quantification is not supported. Some of our analyses introduce negation of facts and run commands of Alloy models. Another example is given in [15, App. B.5.2] where only the negation of a universal quantification makes it solvable by the Alloy Analyzer. In cases where higher-order quantification is present, it could make the analyses impossible using the Alloy Analyzer. Note that our translation never introduces higher-order quantification into Alloy models. A possible solution could be the use of Alloy* [24], an extension of Alloy with higher-order quantification, for analyzing the translation results.

Ordering. Finally, Alloy provides special built-in predicates that *"should not be used in user-level models"*[5]. One particularly useful built-in predicate enables modeling of states and transitions in Alloy and is used in utility model `util/ordering`. However, the use of predicate `totalOrder` leads to unexpected results. As an example, the run command in Lst. 5 produces instances when it clearly should not, i.e., the same predicate evaluates differently on the instance. This limitation is likely to be resolvable with more implementation details for Alloy's built-in predicates.

It is important to note that only the limitation of higher-order quantification can be seen as inherent to our approach while the other to might be seen as implementation issues rather than strict limitations.

[5]See a comment on the use of predicate `totalOrder` by Daniel Jackson at https://stackoverflow.com/a/48748349

Table 1: Model sizes of different collections

Models		loc	#Sig	#Fields	#Fact/fun
Manual	min	1	0	0	1
	median	9	2	1	1
	max	26	5	4	22
Master [2]	min	6	0	0	0
	median	58	4	3	9
	max	383	93	182	124
iAlloy [33]	min	17	1	1	1
	median	47	4	3	6
	max	126	11	8	58
Platinum [36]	min	39	17	8	1
	median	140	49	8	19
	max	231	67	8	25

6 IMPLEMENTATION AND EVALUATION

We have implemented a prototype for the comparison of Alloy specifications on top of the official Alloy implementation in Alloy Tools version 5.10 from [2]. Our prototype does not require any changes to the existing code of Alloy[6].

To evaluate our approach to semantic comparisons of Alloy models, we consider the following research questions:

R1 What is the analysis cost for semantic comparisons?
R2 What is the impact of the limitations of our translation?

Our prototype, the code of our algorithms, and all evaluation materials are available from https://github.com/jringert/alloy-diff.

6.1 Corpus of Alloy Models

We have collected Alloy models from various sources of different sizes and complexity of the employed language features. It is difficult to obtain consecutive versions of Alloy models. However, recent works [33, 36] have collected and shared sets of models with multiple versions.

The corpus for our evaluation consists of a total of 654 Alloy models:

- 53 small, manually created Alloy models exercising most of Alloy language features
- 116 Alloy models of `models-master` from [2] including all 55 specifications from [15]
- 192 Alloy models of specs from the iAlloy benchmark [33]
- 293 Alloy models created for the evaluation of Platinum [36]

We report the sizes of Alloy models in terms of lines of code (of a model, including comments) and numbers of signatures, fields, facts, and functions/predicates (including those from imported models) in Table 1. The overall largest specification is inside the Master [2] collection a model of the firewire protocol [8]. The Platinum [36] collection contains only relatively large models overall. We believe that the combination of these collections provides a diverse corpus for evaluation and analysis.

[6]Our integration into the UI of Alloy requires changes to Alloy's class `SimpleUI`

6.2 Validation

We have validated the prototype on all 654 specifications using different test cases. Basic test cases involve checking the existence of instances of combinations of each model with an empty model and of each model with itself. These test cases have particularly helped to discover earlier bugs in the translation from creating malformed abstract syntax and missing support for language features.

A more complex set of tests semantically compares each model to its lexicographic successor (for models with version histories this is the next version). In this case the comparison goes beyond existence of instances but checks membership in the semantics. As an example for instances of $diff(v1, v2)$ the test case checks whether these are indeed instances of $v2$ and that they are not instances of $v1$. We check this by extending the original models with constraints that formalize the instance in question. Our implementation of instance validation is inspired by ideas from [25, 30].

Some of the 654 models could not be analyzed by our prototype due to the limitations listed in Sect. 5.2. We have manually inspected these models as part of our evaluation in Sect. 6.4 and verified the presence of higher order quantification or ordering in the models.

6.3 R1: Analysis Cost

Our translation from Sect. 4 may, in the worst case, increase the size of each model as detailed in Sect. 4.5. However, the size of the Alloy model might not reveal much about the cost of instance computation. We suggest two measures: the size of the SAT formula and Alloy's running time for finding an instance. To make these measures comparable between different model pairs and sizes we report relative numbers: we divide the measure for $T(v1, v2)$ by the sum of measures of $v1$ and $v2$.

For this analysis we compute satisfiability and the semantic difference in both directions including the first command in each model as described in Sect. 4.4. This covers all computations for the high-level comparisons: refactoring, refinement, extension, and incomparable. We execute the analysis over pairs of consecutive models from version histories. The 654 models contain 408 such consecutive models from collections Master, iAlloy, and Platinum. We run experiments on an ordinary desktop computer with an i7 Intel CPU at 3.7GHz and 64GB RAM running Ubuntu 20.04 LTS with OpenJDK 11. We use Alloy's default options and have selected the SAT solver `CryptoMiniSatJNI` that comes with Alloy (see [34] for a discussion of the use of different solvers).

We measure running times of all processing steps done by Alloy and our prototype (including parsing, transformation, simplification, and solving). The reason to combine all steps rather than restricting it to SAT solving, is that in some cases Alloy's simplifications can determine unsatisfiability without the SAT solver.

The results of our evaluation for computing the semantic difference for all model pairs are shown in Tbl. 2. The columns show aggregated ratios of minimum, median, and maximum over all runs. The rows show ratios for different scopes starting from Alloy's default scope of 3 and increasing in steps of 5 until 18. We compute scopes as in Sect. 4.4, but we cut scopes off at the custom scope, e.g., for custom scope 5 a signature with scope 3 stays at 1, a signature without a scope receives scope 5, and a signature with scope 40 receives scope 5.

Table 2: Relative measures of SAT problem variables and analyses times for growing scope

Models		s=3	s=8	s=13	s=18
SAT problem	min	0	0	0	0
$\frac{\#vars(T(v1,v2))}{\#vars(v1)+\#vars(v2)}$	median	1.5	1.1	1.2	1.9
	max	56	56	225	271
Solving time	min	0.5	0	0	0
$\frac{time(T(v1,v2))}{time(v1)+time(v2)}$	median	1.3	1.4	1.7	2.1
	max	4.5	29.1	29.1	54.8

Not shown in the table are: 6 (7) out of 816 combinations for scope 13 (18) and 13 (13) individual models for scope 13 (18) that took longer than the time limit of 10min. Also filtered out are divisions by 0, when Alloy could determine results by simplification leading to 0 variables in the SAT problem.

Interestingly, the median ratio of variable numbers appears quite stable while the maximum ratio grows more consistently for scopes 13 and 18. In terms of absolute numbers, the maximum number of SAT variables for $T(v1, v2)$ and scope 3 is 31,653 and for scope 18 it is 1,376,866. A similar increase is shown in the ratios of analysis times. Here, the median value ranges from 1.3 to 2.1 and the maximum ratio is at 54.8. Absolute times range from max 3.7 sec at scope 3 to max 6.8 min at scope 18. The maximum total memory consumption including Java, Alloy, and the SAT solver was 780MB for single models, and 4GB for pairs.

While these numbers and their growth look encouraging, it must be mentioned that the increasing analysis cost could become prohibitive for scopes used in commands of some of the example models. Some models in the collection use a scope of 300. However, it is very likely that these scopes are artificial due to the performance oriented nature of the tools iAlloy and Platinum.

6.4 R2: Impact of Limitations

We have identified and discussed limitations of our translation in Sect. 5.2. The ordering limitation (use of predicate `totalOrder`) and the signature kind clash are independent of the analysis kind. However, the limitation of higher-order quantification is dependent on whether a model appears negated or not. To establish worst-case numbers for this limitation, we use the analysis $diff(m, m)$, where model m appears both negated and not. Finally, the limitation of mixed field arities applies to pairs of models. For field arity conflicts, we check consecutive versions of models.

Out of 654 models, 76 (2 higher-order, 73 ordering, 0 field arity, 1 signature kind clash) cannot be analyzed due to limitations described in Sect. 5.2. When taking commands into account (see Sect. 4.4), the number of unsupported cases due to higher-order quantification rises from 2 to 13, which is not surprising as higher-order quantification might be added in the predicate or assertion of the command.

Of the two (13) higher-order problems, one (two) are deliberately crafted in collection Manual and one model is already not analyzable by Alloy, i.e., this inherent limitation has no relevance for the analyzed model collections when ignoring commands. The field arity conflict limitation did not appear once among the analyzed models pairs. The only real limitation in many models (11%) is due

to the use of ordering. Additional nine models failed to be correctly parsed by Alloy and one model failed to analyze due to too many quantified variables[7].

The number of unsupported specifications distributes over the different collections of models as follows: 40 of 116 (34%) from Master [2], 29 of 192 (15%) from iAlloy [33], 0 of 293 (0%) from Platinum [36], and 6 of 53 (11%) from collection Manual. The additional 11 higher-order problems from taking commands into account are one from collection Manual and 11 from collection Master [2].

It appears that collection Master [2] contains the most complex models, e.g., from case studies using Alloy. An analysis of the cases in collection iAlloy [33] reveals that all 29 unsupported models are variants of 7 models from collection Master [2].

6.5 Threats to Validity

Internal. First, our implementation may have bugs. To mitigate, we have validated it as described in Sect. 6.2. We consider our validation on 654 models very strong compared to validation reported in other works. The models used are from different sources and some were created using the mutation tool MuAlloy [31, 32]. Second, any performance analyses involving a Java virtual machine and JNI for native libraries might be subject to running time fluctuations or warm-up effects. To mitigate these effects present aggregated values and also problem sizes for the SAT solver in terms of numbers of variables.

External. Some of the evolution histories of specifications we have used in our evaluation are not evolution steps created by real users. The Platinum [36] and the iAlloy [33] collections include versions of Alloy models created using the mutation engine MuAlloy [31, 32]. These versions might be different from those created by real Alloy users.

7 RELATED WORK

7.1 Evolution of Alloy models

Various works have investigated the evolution of Alloy models. Li et al. [17] studied how beginner users work with the Alloy Analyzer. One of their findings, inspiring a series of work on incremental analysis, is that that changes to models between runs of the Analyzer tend to be small.

Bagheri and Malek [4] presented Titanium for speeding up the analysis of evolving specifications. The work tightens analysis bounds by computing all instances of a previous version in order to improve the computation of an instance of a later version.

Wang et al. [33] apply static and dynamic analyses of changes and solution reuse. Their prototype iAlloy achieves 1.59x speed-up on average [33]. Their analyses of changes is very different from our monolithic translation. It is unclear whether the work could be used for semantic comparisons.

Zheng et al. [36] presented Platinum to remove redundant constraints from Alloy models and reuse previous solutions to reoccurring constraints. They report average improvements of solving times around 66%. Due to the systematic nature of generated constraints in our translation, it is very likely that their constraint optimization algorithm might benefit our work.

Despite dealing with evolution of Alloy models, the motivation of these works and ours is fundamentally different. We evaluate our prototype on the datasets of iAlloy [33] and Platinum [36].

7.2 Analysis of Multiple Models

Guerra et al. [12] analyze product lines of meta-models. Similar to our approach, they combine multiple meta-models into one for analysis using a model finder. Technically this work is very different as it uses CDs with OCL as formalism instead of Alloy. The main conceptual differences are in the lower structural complexity of the meta-model in [12], the handling of inheritance with limitation that we avoid, and the motivation of the work.

Drave et al. [9] presented an approach for analyzing the evolution of feature models. They does not provide an implementation. In addition to previous feature model semantics, Drave et al. [9] suggest an open-world semantics, where added features do not constitute a semantic difference. We consider it an interesting future work to define and evaluate the usefulness of an open-world semantics for Alloy models.

Fahrenberg et al. [11] motivate an approach to semantic differencing, where differences are represented in the modeling language itself. They demonstrated this for Feature Models, automata, and later a variant of class diagrams [10]. Following their motivation, one could say that our work allows for the representation of Alloy model differences as Alloy models.

Finally, Maoz et al. [23] presented CDDiff for the semantic differencing of Class Diagrams via an encoding of two Class Diagrams into one Alloy model. The encoding of [23] relied fully on generic signatures and predicates and did not leverage structural constraints of Alloy as does our translation.

8 CONCLUSION

We have introduced a method that enables the semantic comparison of Alloy models. The supported analyses include semantic differencing and checking for refactoring and refinement. We have demonstrated that it is possible to analyze the semantics of multiple Alloy models within the Alloy Analyzer itself. Our evaluation on 654 models demonstrates the feasibility of the approach in terms of reasonable analysis cost.

We believe that this work has the potential to be useful to many Alloy users and also to developers of model analyses tools that employ Alloy as a back-end.

Finally, our translation is also interesting from a broader modeling language perspective. What are other languages and solvers that can be used for semantic comparison of their own models? Do common transformation patterns exist that can be leveraged for these analyses?

REFERENCES

[1] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John C. Mitchell, and Dawn Song. 2010. Towards a Formal Foundation of Web Security. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010.* IEEE Computer Society, 290–304. https://doi.org/10.1109/CSF.2010.27

[7] See discussion here https://stackoverflow.com/questions/22886819/refactoring-alloy-models

[2] AlloyGithub [n.d.]. Alloy Tools GitHub. https://github.com/AlloyTools. Accessed 5/2020.

[3] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. 2010. On challenges of model transformation from UML to Alloy. *Software and Systems Modeling* 9, 1 (2010), 69–86. https://doi.org/10.1007/s10270-008-0110-3

[4] Hamid Bagheri and Sam Malek. 2016. Titanium: efficient analysis of evolving alloy specifications. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 27–38. https://doi.org/10.1145/2950290.2950337

[5] Hamid Bagheri and Kevin J. Sullivan. 2016. Model-driven synthesis of formally precise, stylized software architectures. *Formal Asp. Comput.* 28, 3 (2016), 441–467. https://doi.org/10.1007/s00165-016-0360-8

[6] Julien Brunel, David Chemouil, Alcino Cunha, and Nuno Macedo. 2018. The electrum analyzer: model checking relational first-order temporal specifications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 884–887. https://doi.org/10.1145/3238147.3240475

[7] Alcino Cunha, Ana Gabriela Garis, and Daniel Riesco. 2015. Translating between Alloy specifications and UML class diagrams annotated with OCL. *Software and Systems Modeling* 14, 1 (2015), 5–25. https://doi.org/10.1007/s10270-013-0353-5

[8] Marco Devillers, W. O. David Griffioen, Judi Romijn, and Frits W. Vaandrager. 2000. Verification of a Leader Election Protocol: Formal Methods Applied to IEEE 1394. *Formal Methods Syst. Des.* 16, 3 (2000), 307–320. https://doi.org/10.1023/A:1008764923992

[9] Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe. 2019. Semantic evolution analysis of feature models. In *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume A, Paris, France, September 9-13, 2019*, Thorsten Berger, Philippe Collet, Laurence Duchien, Thomas Fogdal, Patrick Heymans, Timo Kehrer, Jabier Martinez, Raúl Mazo, Leticia Montalvillo, Camille Salinesi, Xhevahire Tërnava, Thomas Thüm, and Tewfik Ziadi (Eds.). ACM, 34:1–34:11. https://doi.org/10.1145/3336294.3336300

[10] Uli Fahrenberg, Mathieu Acher, Axel Legay, and Andrzej Wasowski. 2014. Sound Merging and Differencing for Class Diagrams. In *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8411)*, Stefania Gnesi and Arend Rensink (Eds.). Springer, 63–78. https://doi.org/10.1007/978-3-642-54804-8_5

[11] Uli Fahrenberg, Axel Legay, and Andrzej Wasowski. 2011. Vision Paper: Make a Difference! (Semantically). In *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6981)*, Jon Whittle, Tony Clark, and Thomas Kühne (Eds.). Springer, 490–500. https://doi.org/10.1007/978-3-642-24485-8_36

[12] E. Guerra, J. de Lara, M. Chechik, and R. Salay. 2020. Property Satisfiability Analysis for Product Lines of Modelling Languages. *IEEE Transactions on Software Engineering* (2020), 1–1. https://doi.org/10.1109/TSE.2020.2989506

[13] David Harel and Bernhard Rumpe. 2004. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer* 37, 10 (2004), 64–72. https://doi.org/10.1109/MC.2004.172

[14] Daniel Jackson. 2002. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.* 11, 2 (2002), 256–290.

[15] Daniel Jackson. 2006. *Software Abstractions: Logic, Language, and Analysis.* MIT Press.

[16] Daniel Jackson. 2019. Alloy: a language and tool for exploring software designs. *Commun. ACM* 62, 9 (2019), 66–76. https://doi.org/10.1145/3338843

[17] Xiaoming Li, Daryl Shannon, Jabari Walker, Sarfraz Khurshid, and Darko Marinov. 2006. Analyzing the Uses of a Software Modeling Tool. *Electron. Notes Theor. Comput. Sci.* 164, 2 (2006), 3–18. https://doi.org/10.1016/j.entcs.2006.10.001

[18] Shahar Maoz, Nitzan Pomerantz, Jan Oliver Ringert, and Rafi Shalom. 2017. Why is My Component and Connector Views Specification Unsatisfiable?. In *20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17-22, 2017*. IEEE Computer Society, 134–144. https://doi.org/10.1109/MODELS.2017.26

[19] Shahar Maoz and Jan Oliver Ringert. 2018. A framework for relating syntactic and semantic model differences. *Software and Systems Modeling* 17, 3 (2018), 753–777. https://doi.org/10.1007/s10270-016-0552-y

[20] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2010. A Manifesto for Semantic Model Differencing. In *Models in Software Engineering - Workshops and Symposia at MODELS 2010, Oslo, Norway, October 2-8, 2010, Reports and Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6627)*, Jürgen Dingel and Arnor Solberg (Eds.). Springer, 194–203. https://doi.org/10.1007/978-3-642-21210-9_19

[21] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. ADDiff: semantic differencing for activity diagrams. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, Tibor Gyimóthy and Andreas Zeller (Eds.). ACM, 179–189. https://doi.org/10.1145/2025113.2025140

[22] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6981)*, Jon Whittle, Tony Clark, and Thomas Kühne (Eds.). Springer, 592–607. https://doi.org/10.1007/978-3-642-24485-8_44

[23] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. CDDiff: Semantic Differencing for Class Diagrams. In *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings (Lecture Notes in Computer Science, Vol. 6813)*, Mira Mezini (Ed.). Springer, 230–254. https://doi.org/10.1007/978-3-642-22655-7_12

[24] Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang, and Daniel Jackson. 2019. Alloy*: a general-purpose higher-order relational constraint solver. *Formal Methods Syst. Des.* 55, 1 (2019), 1–32. https://doi.org/10.1007/s10703-016-0267-2

[25] Vajih Montaghami and Derek Rayside. 2012. Extending Alloy with Partial Instances. In *Abstract State Machines, Alloy, B, VDM, and Z - Third International Conference, ABZ 2012, Pisa, Italy, June 18-21, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7316)*, John Derrick, John S. Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene (Eds.). Springer, 122–135. https://doi.org/10.1007/978-3-642-30885-7_9

[26] Tim Nelson, Natasha Danas, Daniel J. Dougherty, and Shriram Krishnamurthi. 2017. The power of "why" and "why not": enriching scenario exploration with provenance. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 106–116. https://doi.org/10.1145/3106237.3106272

[27] Tim Nelson, Salman Saghafi, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2013. Aluminum: principled scenario exploration through minimality. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 232–241. https://doi.org/10.1109/ICSE.2013.6606569

[28] Tahina Ramananandro. 2008. *Mondex* , an electronic purse: specification and refinement checks with the *Alloy* model-finding method. *Formal Asp. Comput.* 20, 1 (2008), 21–39. https://doi.org/10.1007/s00165-007-0058-z

[29] Matthew Stephan and James R. Cordy. 2013. A Survey of Model Comparison Approaches and Applications. In *MODELSWARD 2013 - Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development, Barcelona, Spain, 19 - 21 February, 2013*, Slimane Hammoudi, Luís Ferreira Pires, Joaquim Filipe, and Rui César das Neves (Eds.). SciTePress, 265–277. https://doi.org/10.5220/0004311102650277

[30] Allison Sullivan, Kaiyuan Wang, and Sarfraz Khurshid. 2018. AUnit: A Test Automation Tool for Alloy. In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*. IEEE Computer Society, 398–403. https://doi.org/10.1109/ICST.2018.00047

[31] Allison Sullivan, Kaiyuan Wang, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. 2017. Automated Test Generation and Mutation Testing for Alloy. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*. IEEE Computer Society, 264–275. https://doi.org/10.1109/ICST.2017.31

[32] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. 2018. MuAlloy: a mutation testing framework for alloy. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 29–32. https://doi.org/10.1145/3183440.3183488

[33] Wenxi Wang, Kaiyuan Wang, Milos Gligoric, and Sarfraz Khurshid. 2019. Incremental Analysis of Evolving Alloy Models. In *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11427)*, Tomás Vojnar and Lijun Zhang (Eds.). Springer, 174–191. https://doi.org/10.1007/978-3-030-17462-0_10

[34] Wenxi Wang, Kaiyuan Wang, Mengshi Zhang, and Sarfraz Khurshid. 2019. Learning to Optimize the Alloy Analyzer. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*. IEEE, 228–239. https://doi.org/10.1109/ICST.2019.00031

[35] Pamela Zave. 2017. Reasoning About Identifier Spaces: How to Make Chord Correct. *IEEE Trans. Software Eng.* 43, 12 (2017), 1144–1156. https://doi.org/10.1109/TSE.2017.2655056

[36] Guolong Zheng, Hamid Bagheri, Gregg Rothermel, and Jianghao Wang. 2020. Platinum: Reusing Constraint Solutions in Bounded Analysis of Relational Logic. In *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12076)*, Heike Wehrheim and Jordi Cabot (Eds.). Springer, 29–52. https://doi.org/10.1007/978-3-030-45234-6_2