Karlsruhe Institute of Technology

Study Thesis

Proving Alloy models by introducing an explicit relational theory in SMT

Jonathan Best

19.12.2012

Department of Informatics
Institute for Theoretical Computer Science

Responsible Supervisors: JProf. Dr. Mana Taghdiri
Supervisors: Aboubakr Achraf El Ghazi

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst zu haben und keine weiteren als die angegebenen Hilfsmittel verwendet zu haben.

Jonathan Best
Karlsruhe, den 19.12.2012

0.1 Abstract

This thesis demonstrates that Alloy problems can be proven by using solvers for Satisfiability Modulo Theories (SMT) while preserving the structure of the original problem. The verification condition of the Alloy problem is translated from Alloy to an equi-satisfiable problem in the language of SMT-LIB. The translation is extended with lemmas and shown to be efficiently provable. Furthermore, we motivate a systematic way for finding these lemmas using a hand-written proof and evaluate the process in a case report. The proof is also included in this work.

Contents

0.1 Abstract v

1 Introduction **1**

1.1 Overview 1

1.2 An Example 1

1.3 Outline 2

1.4 Related work 3

1.5 Background 3

2 Approach **5**

2.1 Translating Alloy 5

2.2 On ordered sets 9

2.3 Baseline problem: FileSystem/oneParent 10

3 COM: 'theorem 1' — a case report **13**

3.1 Overview 13

3.2 How we tried to find lemmas in a systematic way 15

3.3 Observations 15

4 Experiments **19**

4.1 Verifying our approach 19

5 Evaluation **23**

5.1 Conclusion 23

5.2 Secondary finding 23

5.3 Future work 24

6 Appendix **25**

6.1 COM: theorem 1 25

6.2 Proof for COM: theorem 1 26

6.3 Selected axioms of RelSMT 27

6.4 Axioms for an ordered signature B 28

6.5 Proof listing for the lemma 'step 45' 30

Bibliography **31**

1. Introduction

1.1 Overview

The purpose of this thesis is to demonstrate that Alloy [15] models can be proven by using a solver for Satisfiability Modulo Theories (SMT). We[1] introduce a translation of Alloy problems[2] into SMT verification conditions (VC) while preserving the structure of the original Alloy problem in SMT. A translation of this sort is often difficult for automatic solvers to handle. We therefore extend the translation with lemmas and show that these extended translations are efficiently provable. Furthermore, we motivate a systematic way for finding these lemmas.

1.2 An Example

Consider a complex problem written in Alloy. There are no counter-examples and now you are tasked with proving its *check* assertion. Before writing a manual proof, you would probably like to use an automated theorem prover. This is where we start.

The following listing shows the 'FileSystem' problem, which we will use as a baseline to evaluate our approach.

```
abstract sig FSO {
   parent: lone Dir
}

sig File extends FSO {}
sig Dir extends FSO {
```

[1] Although these words are the author's, many ideas are not. Thanks and credit goes to the supervision for the idea of making this thesis; for the axioms *subset*, *join*, transitive closure and more; and for the many corrections, lively arguments and support. Further credit goes to Ulrich Geilmann whose collected knowledge[20] laid out, readily encoded in Java.

[2] Terminology: "Alloy problem" refers to the assumptions and the verification condition of a given theorem, "Alloy benchmark suite" refers to all the problems and their input file.

```
   entries: set FSO
}

sig Root extends Dir {}

fact {
   one Root
   no Root.parent
   FSO = Root + Root.^entries
   all o: FSO, d: Dir | o in d.entries => o.parent = d
   entries = ~parent
}

assert oneParent {
   all o: FSO-Root | one o.parent
}
check oneParent for 8
```

The whole problem is translated into the language of SMT-LIB. The resulting file is to long to cite here, so we only show the translation of the verification condition (the "assert" part). SMT-LIB uses a prefix notation with brackets to separate arguments. After each opening bracket, there is a symbol which stand for an operation or a variable. The first symbol is `not`, because the Alloy Analyzer searcher for a counter-example whereas the SMT solver tries to find if the problem is satisfiable. After that, there is a quantified formula with `o` as the bound variable. The symbol `=>` is the logical implication $\Longrightarrow$. Its left-hand argument is the type constraints taken from the corresponding bounding expression in Alloy. The right-hand argument is the actual body of the quantified formula.

```
(assert
 (not
  (forall ((o Atom))
   (=>
    (in_1 o (diff_1 FSO Root))
    (one_1
     (join_1x2 (a2r_1 o) parent))))))
```

1.3 Outline

This paper is split into five parts. We are obviously not the first to address this field, so we will begin with the status quo and how our work furthers the development of Alloy. Part 2 contains the description of our approach to a relevant detail. In part 3, we explore this approach by example. Chapter 4 shows our experiments and results. We conclude this work with chapter 5, where we evaluate our approach and discuss some of the observations made during our work.

1.4 Related work

Proof automation has been around for a while. Recent work by Blanchette, Paulson and Böhme [8, 9] has shown that SMT solvers contribute to automation by verifying proof goals using Isabelle/HOL. We would like to increase the role of the theorem prover by using it to prove Alloy problems.

Alloy is a powerful system with a descriptive modeling language. Its underlying theory is a relational calculus. The problems look very much like object oriented code and often behave similarly. This contrasts with other input formats[3] which are perceived to be "less easy", for example the input languages for Isabelle.

Alloy's major drawback is the limitation to finite scopes. Several publications have approached this issue before: The project "Prioni" [3] and the more recent "Kelloy" [20, 4] mitigate the issue by translating a given Alloy model into proof obligations of an interactive theorem prover. Another approach called "AlloyPE" [1, 12] demonstrates that Alloy models can be translated to SMT for automated checking. AlloyPE could automatically solve a large number of non-trivial problems which is a promising find. Kelloy on the other hand provides a relational theory in first order logic but usually requires a person to operate it. One result of Kelloy was that sometimes inductive assertions can be proven by providing simple additional assertions: *lemmas*.

In general, lemmas are proven assumptions which are used as intermediate steps towards the goal of a proof. We use lemmas to teach the relationship between two or more functions to the solver. We want to improve the existing concepts on the basis of these two findings, producing a relational theory in SMT that can be checked automatically. To this end, we need to find suitable lemmas and determine when they are needed and how to provide them.

1.5 Background

1.5.1 About SMT and RelSMT

The language of SMT-LIB [6] (abbreviated "SMT") is a language adopting a version of many-sorted first-order logic with equality, universal and existential quantifiers and "let" bindings. Additional symbols can be added by "satellite" theories. The core theory implements propositional logic and equality. The input format is a dialect of the programming language Lisp. SMT features free sorts and uninterpreted functions. This work uses SMT-LIB 2.0 as implemented by Z3. To date, there is no theory that implements relational logic.

Like SAT-solvers before them, SMT-solvers are based on the DPLL algorithm for theories, DPLL(T) [13]. To handle quantified logic, most state of the art SMT solvers, like Z3, CVC3 [7] and CVC4 [5] use the E-matching algorithms, based on the backtracking algorithm introduced by the Simplifier theorem prover [10]. This mechanic incorporates quantifier reasoning with satellite ground decision procedures. Z3, our underlying solver, uses a variant with a so-called "E-matching code trees"-index, inspired by saturation-based theorem provers [11]. Alternative algorithms

[3] Referring to other formats in the formal language domain.

have been suggested [18], but they have not yet been compared to each other in detail.

Z3 is also capable of finding satisfiable instances by model-based quantifier instantiation (MBQI). While we do not examine satisfiable problems, this has proven to be useful for dealing with issues that arise in practice.

RelSMT is our extension to SMT, not by satellite theory, but by writing the additional functions and sorts into the problem file. These additional assertions encode the functions and sorts which are needed to express relational problems. SMT problems need the declaration of the occurring sorts and function symbols and the assertions which these functions and sorts are meant to satisfy. From the perspective of the SMT solver, a sort is a set. In RelSMT, the sorts are meta-sets which contain relations or atoms. It is also the name of the tool, which we built to implement this approach.

1.5.2 About Alloy

Alloy [15] is a modeling language based on a relational logic. This means that Alloy supports typical operations for relations and sets. The logic includes operations for transitive closure and cardinality, as well as quantifiers. Alloy has therefore a higher expressiveness than (pure) first-order logic. Every Alloy problem starts with a signature, which is the Alloy term for "set", or "unary relation".

```
sig Object {}
```

Inside the signature declaration are relation declarations, which can be constrained by multiplicity keywords. For the declaration below, the resulting relation is $contains : Container \rightarrow Object$

```
sig Container {
   contains : Object
}
```

Constraints are declared within "facts" and theorems are expressed within "assert" statements. The instruction "check" is a command to the solver to verify a given theorem. Check commands are always scoped; that means there is an upper bound to how many elements for any given relation are created during the check. The Alloy Analyzer tool transforms the given problem to SAT.

2. Approach

2.1 Translating Alloy

The idea of this translation is to provide a relational theory in SMT. It has been demonstrated [17] how axioms can be used to extend first order logic for specific purposes by introducing axioms, so we hope that this will facilitate our approach. We use the same approach to build a relational theory which simulates the Alloy language. Our logic of choice is the logic of linear integer arithmetic and free, uninterpreted sorts and functions. We chose the logic AUFLIA because it fits our purposes the best. Because the SMT core theory is flat typed, it does not support subtyping or multiplicity. We introduce our own type constraints instead.

2.1.1 Scope and Grammar

RelSMT handles most of the Alloy 4 core. Specifically it handles declarations, formulas, expressions, quantifiers, all of the relational operations, and all of the built-in signatures except `Int`. Integer literals and most integer operations are handled, including cardinality. The operation `sum` is not supported. The only handled module is util/ordering. Other modules are not supported. Comprehensions and sequences are not supported either.

The grammar (fig. 2.1.1) of the handled subset has been taken from Geilmann [14] and adapted for RelSMT.

2.1.2 Method

We try to transform Alloy expressions as closely as possible in a manner that preserves the structure of the given Alloy problem. This means every expression in Alloy should have exactly one corresponding expression in RelSMT. Translating into exactly one corresponding expression is not always possible since we need to guard functions and quantifiers for type safety. Another point is that names are translated so that no named expression will occur as an implicit, unnamed expression in RelSMT. Preserving names literally may not always be possible for syntactic reasons.

```
specification ::= open* paragraph*
path ::= ID / [path]
open ::= "open util/ordering" ID [ [ ref,+ ] ] [ as ID ]
paragraph ::= factDecl | assertDecl | funDecl
        | predDecl | cmdDecl | sigDecl
sigDecl ::= [abstract] sig ID [sigExt] { decl,* }
sigExt ::= extends ref | in ref [+ ref]*
factDecl ::= fact [ID] block
assertDecl ::= assert [ID] block
funDecl ::= fun ID [ decl,* ] : declExpr { expr }
predDecl ::= pred ID [ decl,* ] block
cmdDecl ::= check ref [ for number ]
decl ::= ID : declExpr
declExpr ::= [mult] expr | declRelExpr
declRelExpr ::= declRelExpr? [mult] -> [mult] declRelExpr?
declRelExpr? ::= declRelExpr | expr
mult ::= lone | one | some | set
expr ::= ref | this | none | univ | iden
        | (~ | * | ^) expr | expr binOp expr | ref [ [ expr,* ] ]
        | formula [=> | implies] expr else expr
        | { decl,+ blockOrBar }
        | Int [ intExpr ] | ( expr )
binOp = & | + | - | ++ | <: | :> | . | =>
intExpr ::= number_literal | # expr
        | intExpr (+ | -) intExpr | ( intExpr )
formula ::= expr (= | in) expr | expr in declExpr
        | (no | some | lone | one) expr
        | intExpr (< | > | =< | >=) intExpr
        | (! | not) formula | formula logicOp formula
        | formula [=> | implies] forumal else formula
        | quant decl blockOrBar
        | ref [ [ expr,* ] ]
        | ( formula ) | block
logicOp = || | or | && | and | <=> | iff | => | implies
quant ::= all | no | some | lone | one
block ::= { formula* }
blockOrBar ::= block | | formula
ref ::= [moduleRef] ID | univ
moduleRef ::= [path] ID [ [ ref,+ ] ] /
```

Figure 2.1: The subset of the Alloy language which is handled by RelSMT.

Type declarations in Alloy are called "signatures" and represent sets of atoms. They are translated into constants of the sort Rel1. This applies to both top-level signature declarations sig $A\{\}$, as well as subtypes sig B extends $A\{\}$[1]. Sorts themselves do not differ apart from arity, therefore we introduce additional assertions to satisfy the type properties of their original declaration, e.g. that sub-signatures are mutually disjoint.

Relation declarations are fields of signatures. A declaration of arity N is translated to a constant of the sort Rel$N$. For example, the declaration r in sig $A\{lone\ r : B \to C\}$ has arity 3 and is translated to `(declare-fun r () Rel3)`. The type constraints of a relation declaration are translated to additional assertions. Multiplicity predicates like $lone, one, some$ are explicitly built into RelSMT. Multiplicity constraints are translated to assertions using those predicates.

Expressions are either basic or complex. Basic expressions are explicitly declared relations. Complex expressions are the result of relational operations like union, intersection and difference. We translate a given basic expression e with arity N to a constant e of sort `Rel`N. The relational operation is declared as a function of the appropriate arity and types. Each operation comes with a number of assertions (the axioms) to satisfy the operation's semantics. The complex expression is then mapped to function calls of appropriate operations.

Formulas (or "Constraints" as they are called in Alloy) are formed using the operator in. A formula has the form $(expression)\ in\ (expression)$. We introduce a family of membership functions in_N, the subset functions $subset_N$[2] and the sort `Atom` for for atoms. N denotes the appropriate arity. in_N relates N atoms to an expression of sort `Rel`N. Likewise, $subset_N$ relates two relations of arity N. Formulas can be combined to complex formulas using logical operators or quantifiers. Integer literals are either integer expressions[3] or atomic formulas[4] and can be combined with integer operations. Logical operators, quantifiers and integer operations (except for cardinality) are already built into SMT and utilized accordingly.

Functions and predicates that involve relations are mapped to function symbols and assertions in SMT. Function symbols are declared for atoms and relation sorts. The body of the function is translated to a number of assertions, which are all guarded with expressions to satisfy their type properties. A unary function fun with argument of type T and body B is translated to one or more assertions of the form $\{\forall r : \text{Rel1} \mid r \in T \Rightarrow B\}$. Integer operations are mapped directly if the operation exists in SMT. In the special case of cardinality we introduce a new function and appropriate axioms in RelSMT.

[1] Subtypes are subsets $B \subseteq A$, but they also partition the parent set A.
[2] In RelSMT, there is no concept of *proper* subsets, so `(subset_1 A A)` is always *true*.
[3] `(+ 1 2)`
[4] `(> 1 2)`

Relational operators and their interpretation

The relational operators are interpreted by axioms an operator in terms of formulas. In this case, all axioms are based on the operator *in*. Another choice would be to express similar operators in terms of each other[5]. The axiom for transitive closure is defined using first-order logic only, in a similar way to [21]. Axioms for Alloy-only constructs were built to fit the purpose described in their documentation [15].

2.1.3 Transitive closure

Transitive closure is an important tool to formulate graph-related problems like reachability and aggregation. RelSMT uses an integer-less axiomatization to express the closure properties.

A common approach on writing transitive closure in first-order logic, is to extend the logic with integers and then write down the inductive definition: $R^+ = \bigcup R^i; R^1 = R; R^{i+1} = R^i + R$. However, Z3 will not prove inductive facts [2], even when supported by integer arithmetics.

Instead of using the arithmetic approach, we use the following axioms for an integer-less axiomatization:

1. $\forall R \in \texttt{Rel2} \mid trans(R^+)$
2. $\forall R \in \texttt{Rel2} \mid R \subseteq R^+$
3. $\forall R_1, R_2 \in \texttt{Rel2} \mid R_1 \subseteq R_2 \wedge trans(R_2) \Rightarrow R_1^+ \subseteq R_2$

with $trans(\diamond)$ defined as

$\forall R \in \texttt{Rel2} \mid trans(R) \Leftrightarrow \forall a_1, a_2, a_3 \in \texttt{Atom} \mid \langle a_1 a_2 \rangle \in R \wedge \langle a_2 a_3 \rangle \in R \Rightarrow \langle a_1 a_3 \rangle \in R$

Dealing with the power set and completeness

The axioms for transitive closure have some issues, which need to be discussed. In particular, they are not complete. That means that not all valid instances (which satisfy these axioms) have a proper transitive closure. There is a very simple counter-example found by Z3 in small problems:

R_1	$R_2 \cong R_1^+$
$\langle 2\ 3 \rangle$	$\langle 0\ 1 \rangle$
$\langle 3\ 4 \rangle$	$\langle 3\ 4 \rangle$
	$\langle 2\ 3 \rangle$
	$\langle 2\ 4 \rangle$

This false counter-example shows two relations that satisfy the axioms introduced above, yet R_2 is not actually the transitive closure of R_1.

The problem of this counter-example is, that some relations from the power set of R_1^+ are missing. For our axioms to work, we need to make sure that the solver considers the power set of the relation under the transitive closure operator.

[5]For example, the operator *disjoint* could be expressed in terms of $\bigcap$ and $\emptyset$.

Although many operators are supported by RelSMT, the translated problem will not contain them all. In order to provide an efficient translation, it was our goal that operators should only be included, if they occur in the original problem.

In order to assert the existence of the power set for the problems at hand, some of the axioms are always included whether they occur in the original problem or not. Those axioms are $\emptyset, \times, \subseteq, \cup$, the conversion function $\texttt{a2r}_x$ for Atoms to Relations and yet another equality axiom $\forall r, s : \texttt{Rel}_x | r = s \Leftrightarrow (\forall a_{1:x} : \texttt{Atom} | \langle a_{1:x} \rangle \in r \Leftrightarrow \langle a_{1:x} \rangle \in s)$[6] for arity 1 and 2. We found that these are sufficient to prevent false counter-examples. [7]

2.1.4 On the equality of expressions

Equality is expressed with the = symbol in SMT. This symbol can mean the logical connective 'if and only if', if the arguments are boolean, and 'identity' otherwise. The two interpretations could sufficiently express all formulas of our theory, but within RelSMT there is also the interpretation of equality as mutual inclusion: For a given term $A = B$ we translate the expression $(A \subseteq B) \wedge (B \subseteq A)$ instead. We use this mutual inclusion to express all equality statements, where the arguments are relations. For formulas, atoms and integers, we use the built-in [8] equality operator. This decision has some practical implications. Using mutual inclusion instead of built-in equality enables us to solve some problems which we could otherwise not handle at all. An example is the "theorem 1" problem of the COM benchmark-suite, which we will examine in chapter 3.

On the other hand, this mutual inclusion can have a negative impact on some models. During exploration, we found that problems with long runtimes would typically not terminate until much later or not at all[9]. This can be discouraging, because intermediate results are hard to find. We suggest that during exploration, the built-in equality operator should be used for all expressions. In those cases, we used an appropriate lemma instead. The lemma simply states the mutual inclusion as an implication: $A = B \implies (A \subseteq B) \wedge (B \subseteq A)$. For the results presented in this work, we used relational equality, unless otherwise noted.

2.2 On ordered sets

Alloy problems can be extended with modules. Modules are a powerful way of increasing capabilities while keeping the language simple. Alloy comes with some commonly used, built-in modules. One of these modules is the ordering module. We added support for the documented operations in order to examine more complex models. The actual examination didn't make it into this paper and has to be addressed in future work. Other modules apart from the ordering module are not supported.

When discussing order in set theory, it can be useful to distinguish between finite and infinite cases. We define order as a function of integer and employ SMT's

[6] Introduced in [19]

[7] They may not actually be sufficient to assert the existence of the power set in arbitrary cases.

[8] We will refer to SMT's = operator as 'built-in equality'.

[9] See (3.3.4)

integer arithmetic to properly instance it. Order is always declared for a Signature. We define a function *ord* which maps each element of a signature to an integer. We also define a function *at* which stands for "element at given position". *at* maps each Signature and integer to an element. We also define that two elements at the same position must be equal, so every ordering in RelSMT is total. The operations next, previous, max, min and cardinality are defined in terms of *at* and *ord*. Their definitions are listed in (6.4).

An ordering in RelSMT is a bijection of Signature-elements and integers. We distinguish the cases of finite Signatures and possibly infinite signatures. Since this distinction is not supported in Alloy, we let the user declare finite signatures by command line switch. If a signature is a-priori assumed to be finite, we define the functions for maximum and cardinality. If the user has made not declared finiteness, no assumption is made. We feel that this is a sensible default, because the problems that we examined make that assumption as well, i.e. they model "real-world" systems which are typically finite.

In application, the decision whether a Signature should be finite, is left to the solver. If RelSMT encounters an ordered Signature, it will generate the supported functions and guard them with the uninterpreted function `finite`. If the solver finds that `(finite A)` should be *true*, functions like `(last A)` will have defined values. For example, the function `last` is defined as $\emptyset$ if the Signature is not finite, and denoting the last element of an ordered Signature if finite.

The assumption that signatures are infinite in general would also cause issues with satisfiable problems. Problems arise, if none of its instances was finitely describable or Z3 fails to find a finite description. Consider a problem which is known to be *sat* and the only satisfying instance has an infinite domain. If Z3 attempts to find an instance satisfying this problem, it has to interpret this infinite domain. Since Z3 can only find interpretations which are finitely describable, it may not be able to solve the problem.

2.3 Baseline problem: FileSystem/oneParent

We examine the baseline problem line by line. The problem describes a simple file system with files and folders (or "directories"). Also, all the objects in the file system share a common interface.

The basis for a file system is the file system object. From a programming perspective, this is like an interface or an abstract class. The keyword `abstract` means that the subsets of `FSO` for a full partition. `sig` declares a "signature", which is just another word for "set" or "unary relation".

```
abstract sig FSO {
```

The following line declares the binary relation $parent \subseteq (FSO \times Dir)$. It is constrained by the keyword `lone` so that each `FSO` can have at most one corresponding `Dir`. From the programmers perspective, this relation behaves like a nullable reference to a single folder.

```
    parent: lone Dir
}
```

These two lines declare files and folders. The keyword `extend` constrains the two signatures to be subsets of `FSO`.

```
sig File extends FSO {}
sig Dir extends FSO {
```

This line declares the relation $entries \subseteq (Dir \times FSO)$. The keyword `set` removes any constraints from the codomain. It meanst that the join expression `Dir.entries` behaves like a set.

```
    entries: set FSO
}
```

This line declares the signature `Root`, which will be constrained to a single atom later on.

```
sig Root extends Dir {}
```

The keyword `fact` declares a block of arbitrary constraints. They are assumptions about the problem.

```
fact {
```

There can only be a single atom in the signature `Root`.

```
    one Root
```

The Root is not in the domain of `parent`. The right-hand expression of `no` yields the empty set.

```
    no Root.parent
```

The following line means that all file systems objects can only be either the root or a descendant of the root. `^entries` denotes the transitive closure of `entries`. So `FSO` is constrained in the following way: The whole signature `FSO` contains `Root`, unified with the transitive closure where the domain is (in) `Root`.

```
    FSO = Root + Root.^entries
```

If a file system object is in the entries of a directory, the objects' parent must be this directory, for all file system objects `o` and directories `d`.

```
    all o: FSO, d: Dir | o in d.entries => o.parent = d
```

`entries` is the transposition of `parent`.

```
    entries = ~parent
}
```

The keyword `assert` denotes the theorem which is up for evaluation. From our point of view, this is the verification condition of the examined problem.

```
assert oneParent {
```

The binary operator − is the set difference. The oneParent problem verifies that each file system object `o` has exactly one parent, except for the root.

```
    all o: FSO-Root | one o.parent
}
```

The keyword `check` is a command to the Alloy Analyzer to find a counter-example. It gives the name of the assertion `oneParent` and the scope `8`.

```
check oneParent for 8
```

3. COM: 'theorem 1' — a case report

3.1 Overview

In order to verify our theory, we wanted to explore the concept on a single sufficiently interesting problem. We have selected a theorem of the Component Object Model for this purpose. At the same time, the theorem is complex in terms of syntax and, for its real-world application, is easy enough for a student to understand and prove by hand. In our case, a proof could be found within two work days[1].

The Component Object Model (COM) is a specification of "an infrastructure for the creation, operation, and management of components"[16]. The problem has been subject to several years of analysis and its theorems are widely believed to be correct. In our attempt for a structured search we contribute our own proof (see 6.2) for the first theorem of the benchmark suite. The full Alloy problem is printed in appendix (6.1).

The syntactic complexity of the "theorem 1" problem is very high compared to our baseline, "filesystem". In this case, complexity means "many different constructs". The COM benchmark contains several interesting concepts:

- There are three primary signatures and two sub-signatures representing IDs, interfaces and components;
- The ternary relation "qi"[2] relating two Interfaces by means of an ID;
- Equivalence among interfaces defined by reflexivity, symmetry and transitivity;
- A fact using transitive closure to express "aggregation" among components

An excerpt (fig. 3.1) from the model shows how the central signature `Interface`. The definition contains the relation `qi` as well as two other important declaration. The relation `iidsKnown` constrains the domain of `qi` and `reaches` constrains the range of `qi`.

[1] That means that the total time spent on the proof was approximately 16 hours.
[2] This relation models the "query interface" function.

```
sig IID {}
sig Interface {
  qi : IID -> lone Interface,
  iids : set IID,
  iidsKnown : set IID,
  reaches : set Interface
}
fact {
  all i: Interface |
    (i.iidsKnown = i.qi.Interface) and
    (i.reaches = IID.(i.qi))
}
```

Figure 3.1: The basic signatures of COM: the interface and its query function `qi`.

The semantic complexity is within a manageable scope. The proof for this theorem has multiple steps[3], including a case distinction. Therefore, the theorem is not trivial. However, no induction is needed to prove the theorem. This makes it manageable for an automated theorem prover.

The model is primarily concerned with nested structures. It contains five theorems which claim properties about reachability (1), the hiding interfaces (2, 4a) and the equality of components (3, 4b). During the case study we examine only the first theorem:

> "*Theorem 1* says that for any legal component, the identifiers known to any interface of that component are all the identifiers of the component. In other words, every interface identifier of a component is accessible from every interface of that component." [16]

```
assert Theorem1 {
   all c: LegalComponent | all i: c.interfaces | i.iidsKnown = c.iids
}
```

Figure 3.2: Verification condition of COM:theorem 1: *"Every iid a component exports is known to all its interfaces."*

The theorem claims that two differently constructed sets hold the same elements. The sets are `i.iidsKnown`, IDs known to an interface, and `c.iids`, the IDs which are registered with a component. The benchmark contains facts that easily show that `c.iids` is a subset of `c.iids`, but the other way around has to be inferred. During our study we could easily show that a proof for $i.iidsKnown \subseteq c.iids$ is easily found by Z3, but not the other way around.

[3]Shorter proofs exist, because the Alloy Analyzer is able solve the problem using fewer facts than we have used in our proof. However, we did not find any shorter proofs.

3.2 How we tried to find lemmas in a systematic way

For lack of a better starting point, we used a hand-written proof[4]. This road-map lays out the strategy to our analysis:

1. Write a proof of the theorem inside the check-command's assertion by hand.
2. For each step S of the proof, write down all the assumptions, including the facts from the benchmark and all previous steps. Keep the proof-steps $\{R\}$ separated[5] from the facts $\{F\}$.
3. Let the current step be S, the previous step be R and the facts be $\{F\}$. To verify an inference step $\{F, R\} \vdash S$, set up Alloy with a benchmark consisting of facts $\{F, R\}$ and check $\{S\}$. To verify a material equivalence[6] $R \Leftrightarrow S$, set up Alloy with a benchmark consisting of facts $\{F\}$ and check $\neg \{R, S\}$. To verify a corollary $\vdash S$ (which follows directly from the facts) set up a benchmark consisting of facts $\{F\}$ and check $\{S\}$.
4. If Alloy finds no counter-examples, proceed.
5. Invoke Alloy2RelSMT and check the result with Z3.
6. Z3 should return UNSAT. If does not finish at all, the step is too hard. If the step can be expressed in a general form, it may be a candidate for a lemma.
7. Write a lemma for each step in RelSMT if possible. Start with the corollaries. Add it to the RelSMT theory.
8. Go to 5.
9. For each step S repeat 3 but replace R with its preceding step R'. The result of this step shows if Z3 is able to 'skip' steps in the proof.

The author found that the distinction between hard and easy steps did not yield the expected results. A step being "hard" does not predict the need for a lemma. A step being "not hard" does not predict the solver's ability to solve several such steps in a row. In fact, most of the steps previously identified as "hard" were solvable after "minifiying" the benchmark (see 3.3.5). Four lemmas were written for steps that were solvable in step 5. One of them turned out to be useful and is discussed later.

3.3 Observations

3.3.1 What did work, what didn't?

As a proof strategy we choose proof by contradiction. The verification condition (VC) posits the relational equality of two sets (expressions with arity 1). We make a case distinction and get two paths in the proof, both of which must be led to a contradiction. We will call them the "hard" path expression (3.3) and the "easy" path expression.

[4] Appendix (6.2)

[5] "Seperate" as in thinking about them as two different kinds of steps.

[6] The expression 'R holds if and only if S holds'

(A2) **assume** $\exists c \in LegalComponent \mid \exists i \in c.interfaces \mid c.iids \not\subseteq i.iidsKnown$

Figure 3.3: The "hard" path expression.

The solver can derive all the steps for the easy path directly from the model without any additional lemmas. These "simpler" steps follow by term substitution from the model and do not require interpretation of the model. However, the steps are not sufficient to disprove the false assumption.

The "hard" steps do not easily follow from the facts alone. To obtain these steps, we used relational reasoning and set theory.

3.3.2 The lemma we found and how effective it was

When exploring this problem, it wasn't clear what a lemma should express. A first idea was to daisy-chain the steps of the hand-written proof together by lemmas and hope that the solver could somehow end up with the same conclusion as we did. This turned out as not being very effective. The solver is pretty good when it comes to this type of approach. In fact, this is probably how Z3 proved the easy path, by substituting compatible expressions from one term for another. Examining the steps of the proof, we noticed that the hard steps involved reasoning beyond the syntax. Specifically, we used the relationship between set inclusion and the *join* operator, which had not been a part of RelSMT until we wrote down the corollary and included it as a lemma. This kind of rational could not be expressed with a daisy-chain.

We dubbed this corollary 'step 45'[7] (fig. 3.4). with reference to the step in the proof (6.2) where we introduced it. The lemma itself is provable by the solver and an appropriate benchmark for this proof is included in appendix 6.5.

After appropriate modification [8], this corollary was identified as the only needed lemma for this theorem. It states that given a relation and its subset (e.g. $\in$ `Rel1`), the results of joining each with the same expression (e.g. $\in$ `Rel2`) must be subset as well.

$$\forall\ a, A \in \text{Rel1}, R \in \text{Rel2} \mid (a \subseteq A) \Rightarrow (\ a.\,R \subseteq A.\,R\ )$$

Figure 3.4: Step 45, the lemma for COM: 'theorem 1'

3.3.3 What we learned and how to write a lemma

Solving the hard parts can get very cumbersome. In early trials we set up Z3 to prove that one "hard" step follows from another. We observed that showing this for consecutive steps is possible, but takes a long time and uses a lot of memory.

[7] There are also other lemmas which were found by chance. An initially wrong approach to the proof yielded many different steps that could easily be "daisy-chained" with lemmas. Although they occasionally seemed to have an effect on other problems, those observations were not reproducible and are therefore not included.

[8] See 3.3.5 on the exclusion of facts

A different approach: Ground terms

When the Z3 solver attempts to refute a given quantified formula, it starts by fitting ground terms against the expression [11]. If the proposition contradicts a ground term, the solver stops. If no match can be found, it begins instancing quantifiers with its ground terms and checks the theorem again. It will continue this cycle, until a contradiction has been found.

The author has learned during that a hand-written proof should reflect this behavior. For E-Matching-based solvers like Z3, CVC3 and CVC4, such a proof could reveal more useful lemmas than the ex-falso proof shown in this paper. A lemma should be written in such a way that instancing it with a ground term yields a step in the proof. However, this idea is not explored in this work.

3.3.4 The effects of equality representation on performance

During our manual proof, we split the verification condition into two inclusions to show equality. This is the textbook-approach taught in many first-semester lectures on set theory.

Since our theory did not initially have the concept of mutual inclusion, the solver had no starting point to follow our proof. For the purposes of discovery, we changed the verification-condition from an equality-expression to a mutual inclusion. This alone did not have any visible effect, so we tried to find other equalities which did. We found such equality at "line 41"[9] from COM: 'theorem1', the fact where the domain of 'qi' is defined. The line itself does not affect the problem in terms of validity and is not used in our proof. However, it does have an effect on performance. Changing both the VC and "line 41" enabled Z3 to come to a conclusion for the first time during this study.

```
fact {
  all i: Interface |
    (i.iidsKnown = i.qi.Interface) and    // this is line 41
    (i.reaches = IID.(i.qi))
}
```

Figure 3.5: "line 41" and its surrounding context

These observations led to the design decision for expressing equalities as mutual inclusions by default. However, always using mutual inclusions does not produce the same results. Generally speaking, it increases the runtime for all problems. We also found that changing all the operators does not reproduce the results from above, even with timeout limits of an hour and above.

3.3.5 How the exclusion of certain facts sped up the solution

When exploring, we usually did not work with the whole benchmark. Many of the facts are irrelevant to the theorem which we are trying to prove. We selected facts

[9] Due to difficulties in document management, this is just a denomination, not an actual line number.

[10] Spurious result.

	”line 41” as built-in	”line 41” with relational equality
VC as built-in	timeout (5 m)	timeout (5 m)
VC as split into two clauses of mutual inclusion	timeout (5 m)	unsat (31 s)
VC as a single clause with relational equality	timeout (5 m)	unsat[10] (25 s)

Table 3.1: Effects of relational equality on single expressions

at random and removed them from the problem. If Alloy's finding of the smaller problem was consistent, we used it. However, we did not remove any facts that were used in the proof, whether they had been relevant or not.

This led to some interesting observations. For one, the problem now terminates, if the VC is expressed as a mutual inclusion. It also terminates, if there is a lemma which allows the rewrite of the equality operator as a mutual inclusion.

From the viewpoint of a logic problem, this resembles proving only the assertion's core. At this stage of the exploration, it is good enough to work with the core alone. The problems which were modified in this way, are called ”minified” and marked with an asterisk *.

4. Experiments

4.1 Verifying our approach

To verify the effectiveness of our approach, we design an experiment with several steps. Unfortunately, this experiment turned out to be too ambitious for the available time. Therefore, only "Theorem1" was properly analyzed and will be evaluated.

In each step, we translate a particular set of well-known problems and run the resulting verification condition with the Z3-solver. The response of Z3 is the result of the step and may be "sat", "unsat" or "unknown"[1]. We then choose a problem with unknown result and try to find general lemmas about our operators which lead Z3 to yield the correct response. After adding the new lemmas to RelSMT we begin a new step and translate all problems again and determine if the changes to the lemmas carried over to other problems. Our initial problem will be the check "oneParent" of the model "FileSystem".

"FileSystem" encodes a simple, abstract file system found in modern operating systems. It asserts some basic properties about the system's consistency and is suitable used for teaching Alloy. The check "oneParent" asserts that every file system object, except for the root, has exactly one parent object that is its parent. "someDir" asserts that directories exist and "fileInDir" every file is inside a directory. "MarksweepGC" is a model for the basic mark and sweep garbage collection algorithm. "Soundness1" asserts that the algorithm leaves marked objects untouched. "Soundness2" verifies that no object is both marked and unreachable. "Completeness" is the property that all objects that have not been marked are unreachable as well.

All problems have been taken from the Alloy book [15] and the sample library. None of the chosen problems can be solved without lemmas. We use this list of problems for our experiment:

[1] For the purpose of this experiment, the result "unknown" also includes time-out.

Problem	Description	Check	N^o
FileSystem	Model of a generic file system	oneParent	1
		someDir	2
		fileInDir	3
MarksweepGC	Model of mark and sweep garbage collection	Completeness	4
		Soundness1	5
		Soundness2	6
COM	COM interface and aggregation mechanism	Theorem1	7
		Theorem2	8
		Theorem3	9
		Theorem4a	10
		Theorem4b	11

4.1.1 Baseline for our experiments

The baseline test (table 4.1) is the starting point of the experiment. Whenever a new lemma is introduced, its effectiveness is measured relative to this test. This basic set of lemmas contains one lemma which deals with transitive closure and a family of lemmas pertaining to the join-Operation (fig. 4.1).

As you can see in table 4.1, there are two problems that we can solve using only the baseline: FileSystem/oneParent and MarksweepGC/Soundness1. The lemmas were developed for oneParent, but they also affect Soundness1. We had hoped to observe this behavior during other iterations as well. If a lemma causes this effect, we say the lemma 'generalizes' to other problems.

```
; weak lemma 1 for transClos about the second-last 'middle element'
(assert (!
  (forall ((a1 Atom)(a3 Atom)(R Rel2)) (=>
    (in_2 a1 a3 (transClos R))
    (exists ((a2 Atom)) (in_2 a2 a3 R))))
  :named lemma6816308a ))

; 2. lemma for join_1x2. direction: in to join
(assert (!
  (forall ((a1 Atom)(a0 Atom)(r Rel2)) (=>
    (in_2 a1 a0 r)
    (in_1 a0 (join_1x2 (a2r_1 a1) r))))
  :named lemma6ec6a62 ))
```

Figure 4.1: Two exemplary lemmas from the baseline problem.

[2]Minified version of the original problem. See (3.3.5).

Problem	Check	N^o	Result	Time/Timeout
FileSystem	oneParent	1	unsat	0.01
	someDir	2	-	550.00
	fileInDir	3	-	550.00
MarksweepGC	Completeness	4	-	550.00
	Soundness1	5	unsat	0.09
	Soundness2	6	-	550.00
COM	Theorem1*[2]	7	-	550.00
	Theorem2	8	-	550.00
	Theorem3	9	-	550.00
	Theorem4a	10	-	550.00
	Theorem4b	11	-	550.00

Table 4.1: Baseline results for the Alloy problems

4.1.2 First iteration: Extending with lemma found for COM: theorem1

The lemma for "step 45"(3.4) had no visible effect on any other problem. Compared to the baseline, only theorem1 is now solved. We can say that this lemma did not generalize.

Problem	Check	N^o	Result	Time/Timeout
FileSystem	oneParent	1	unsat	0.01
	someDir	2	-	550.00
	fileInDir	3	-	550.00
MarksweepGC	Completeness	4	-	550.00
	Soundness1	5	unsat	0.07
	Soundness2	6	-	550.00
COM	Theorem1*	7	unsat	5.00
	Theorem2	8	-	550.00
	Theorem3	9	-	550.00
	Theorem4a	10	-	550.00
	Theorem4b	11	-	550.00

Table 4.2: Results for the Alloy problems after adding the lemma found for COM: theorem1.

4.1.3 More iterations

We did not have the time to evaluate any problems besides oneParent and theorem1. The experiment was designed for up to ten iterations. We felt that this number was large enough to observe lemmas found in one problem affecting another. However, every iteration includes writing a proof by hand, finding a lemma within the proof and testing the new lemma against the other problems. Therefore, the majority of

the problems remains not evaluated. They are listed for the sake of accuracy and completeness.

5. Evaluation

5.1 Conclusion

In the current paper we attempted to show that using an abstract theory with a SMT-solver is possible. We found that this abstract theory was not very powerful if left in an unrefined state. There were few examples which were solvable without modification. We needed lemmas to make the usage of our theory feasible. These allowed the solver to take shortcuts in its search to find a proof quickly. However, we could not show that our approach was generally applicable. In order to proceed with our experiments, it was necessary to strip the examined problems to their core.

The clauses which we removed during preparation consisted more or less of large quantified formulas. We therefore suspect that adding all-quantified clauses to a problem can lead to issues, even in the presence of shortcuts. We suggest that the approach presented in this paper will not be enough. We expect that the step of adding lemmas should be followed by another step of subsuming lemmas. This may be necessary until solvers get rapidly better at finding unsatisfiable subsets.

Furthermore, we introduce a method for finding new lemmas in a structured way. We posit that this method can be used to "teach" previously unsolvable proof steps to the solver. In our case study the solver succeeded by using the lemma found with this method. Therefore we suggest that this method may be used to explore new lemmas. The question remains whether its benefit outweighs its significant cost.

We had hoped to show that our lemmas generalize across multiple problems. So far, only the ones for our baseline test applied to more than one problem. The systematically found lemma did not apply to any other.

During our research, we briefly examined the possibility of extending our translation to other Alloy features. We introduced an application for Alloy problems that uses the module for ordered sets, but did not examine it.

5.2 Secondary finding

During the development of our tools, there were some obstacles to overcome. Since it is not feasible to attach a debugger to Z3, we had to find other ways of showing

intermediate results. This attempt on extracting cores from the solver is submitted for the benefit of the reader.

5.2.1 On constructing cores

Lemmas should be shortcuts for the solver to reach a finding. This finding does not necessarily mean just *sat* or *unsat*. We have observed a scenario where adding lemmas helped constructing a core. We started with a set of unsatisfiable clauses $\{c_1..c_n\}$ which the solver identified as *unsat*. However, the solver did not succeed in constructing a core[1]. After adding four lemmas $c_{(n+1)}..c_{(n+4)}$ to the set, the solver managed to produce a core in a timely manner.

To our surprise, the core consisted only of clauses from the original set $\{c_1..c_n\}$. So, despite helping the solver reach a conclusion, the lemmas did not have an actual influence on the solver's conclusion.

During a third run, we removed all clauses which were not part of the core. This resulted in a set $c_r..c_s[1 < r < s < n]$, which did not include our new lemmas. The solver managed both to find these clauses *unsat* and to produce a core. This goes to show that adding lemmas to the original problem helped the solver identify "irrelevant" clauses. By removing the clauses which were not part of the core, we effectively saved the solver the trouble of dealing with them.

5.3 Future work

We would like to extend our translation system even further. At the moment, only core features of Alloy are supported. Especially set comprehensions are not, though it should be possible, as shown in [17] for Spec♯. It should be interesting to implement these for SMT and examine their behavior.

We continue to hold the expectation that lemmas can generalize across problems. Since only very few examples were examined during this work, we have no basis to reject this expectation. Further research should examine a larger set of problems with respect to single lemmas. Likewise, our systematic method for finding these has not been tested on more than the one example and needs empirical examination.

While expanding to a larger set of examples, the negative impacts of additional lemmas should also be considered. Some lemmas are costly in terms of memory and runtime, thus a system for selecting appropriate lemmas may be needed. For example, given sufficient experimentation time, this system could be devised by testing all permutations of applicable lemmas.

[1] By definition, the core of a set of clauses is the smallest unsatisfiable subset. However, Z3 does not guarantee to find the minimal subset; hence we say 'a core' and not 'the core'.

6. Appendix

6.1 COM: theorem 1

This is the full Alloy problem for "COM: theorem1" as we evaluated it during the experiment. The comments mark the lines where we removed or modified lines in the original benchmark.

```
module exploration/com1_minified

sig IID {}

sig Interface {
  qi : IID -> /* lone */ Interface,
  iids : set IID,
  iidsKnown : IID,
  reaches : Interface
}

fact { all i :Interface |
                              // line intentionally left blank
  (i.reaches = IID.(i.qi))  // reaches = ran[qi]
}

sig Component {
  interfaces : set Interface,
  iids : set IID,
  identity : interfaces,
}

fact IdentityAxiom {
  some unknown : IID | all c : Component |
    all i : c.interfaces | unknown.(i.qi) = c.identity
}
```

```
fact ComponentProps {
  all c : Component | c.iids = c.interfaces.iids
}

sig LegalInterface extends Interface { }
fact { all i : LegalInterface | all x : i.iidsKnown | x in x.(i.qi).iids}

sig LegalComponent extends Component { }
fact { LegalComponent.interfaces in LegalInterface }

fact Symmetry { all i, j : LegalInterface | j in i.reaches => i.iids in j.iidsKnown }

fact Reflexivity { all i : LegalInterface | i.iids in i.iidsKnown }
fact Transitivity { all i, j : LegalInterface | j in i.reaches => j.iidsKnown in i.iidsKnown }
// removed the fact for aggregation

assert Theorem1 {
    all c: LegalComponent | all i: c.interfaces | c.iids = i.iidsKnown
    }

check Theorem1 for 5 expect 0
```

6.2 Proof for COM: theorem 1

The following proof was used to find a suitable lemma for theorem 1. The numbers in front of the steps are arbitrary. The formulas are written in Alloy. The steps beginning with "some" usually follow from one or more previous steps. Steps beginning with "all" are taken directly from the benchmark or derived from the benchmark.

Proof outline for disproving the first case:
$(((((A1 \wedge 8) \Leftrightarrow 9) \wedge 10 \wedge (11 \Rightarrow 12)) \Rightarrow 13) \wedge 14) \Leftrightarrow 15 \perp$

Proof outline for disproving the second case:
$((A2 \wedge 14) \Leftrightarrow 42) \wedge 45) \Rightarrow 43) \Leftrightarrow 44 \perp 38$

theorem all c: LegalComponent | all i: c.interfaces | i.iidsKnown = c.iids

1 assume some c: LegalComponent | some i: c.interfaces | i.iidsKnown != c.iids

2 some c: LegalComponent | some i: c.interfaces | (i.iidsKnown not in c.iids) or (c.iids not in i.iidsKnown)

3 some c: LegalComponent | some i: c.interfaces
| (some o: i.iidsKnown | o not in c.iids)
or (some o : c.iids | o not in i.iidsKnown)

case-by-case:

case A1 ... (o in i.iidsKnown), based on 3

A1 assume some c: LegalComponent | some i: c.interfaces |
(some o: i.iidsKnown | o not in c.iids)

8 all i : LegalInterface | all x : i.iidsKnown | x in x.(i.qi).iids

9 some c: LegalComponent | some i: c.interfaces | some o: i.iidsKnown | o in o.(i.qi).iids and o not in c.iids

10 all i : c.interfaces | all x : IID | x.(i.qi) in c.interfaces

11 sig Interface { iidsKnown : IID }

12 all c: LegalComponent | all i : c.interfaces | all o : i.iidsKnown | (o in IID)

13 some c: LegalComponent | some i: c.interfaces | (some o: (c.interfaces).iids | o not in c.iids)

14 all c : Component | c.iids = c.interfaces.iids

15 some c: LegalComponent | some i: c.interfaces | (some o: (c.interfaces).iids | o not in (c.interfaces).iids) $\lightning$

□

case A2 ... (o in c.iids), based on 2

A2 assume some c: LegalComponent | some i: c.interfaces | c.iids not in i.iidsKnown

14 all c : Component | c.iids = c.interfaces.iids

45 all c: Component| all i:c.interfaces | i.iids in c.interfaces.iids

42 some c: LegalComponent | some i: c.interfaces | c.interfaces.iids not in i.iidsKnown

43 some c: LegalComponent | some i: c.interfaces | i.iids not in i.iidsKnown

44 some i: LegalInterface | i.iids not in i.iidsKnown

38 all i: LegalInterface | i.iids in i.iidsKnown $\lightning$ 44

□

■

6.3 Selected axioms of RelSMT

Following is a list of axioms written in SMT. They are all declared with respect to the function in_1 and in_2, which are uninterpreted. These axioms are generated by examining the arity and operations of the input expressions. Each axiom is placed into an assertion and decorated with a :named directive. The assert commands are omitted here.

```
; axiom for set product
(forall ((y0 Atom)(x0 Atom)(A Rel1)(B Rel1)) (=
   (in_2 x0 y0 (prod_1x1 A B))
   (and (in_1 x0 A) (in_1 y0 B))))

; subset axiom for Rel2
(forall ((x Rel2)(y Rel2)) (=
   (subset_2 x y)
   (forall ((a0 Atom)(a1 Atom)) (=>
```

```
      (in_2 a0 a1 x)
      (in_2 a0 a1 y)))))

; axiom for the operator & (disjoint)
(forall ((A Rel1)(B Rel1)) (=
   (disjoint_1 A B)
   (forall ((a0 Atom)) (not
      (and
         (in_1 a0 A)
         (in_1 a0 B))))))

; axiom for the operation . (join between a binary and a unary relation)
(forall ((A Rel2)(B Rel1)(y0 Atom)) (=
   (in_1 y0 (join_2x1 A B))
   (exists ((x Atom)) (and
      (in_2 y0 x A)
      (in_1 x B)))))

; axiom for 'the expression is empty'
(forall ((a0 Atom)(R Rel1)) (=>
   (no_1 R)
   (not (in_1 a0 R))))

; axiom for the empty set
(forall ((a Atom)) (not (in_1 a none)))

; axiom for the conversion function for Atoms to binary relation
(forall ((x0 Atom)(x1 Atom)) (and
   (in_2 x0 x1 (a2r_2 x0 x1))
   (forall ((y0 Atom)(y1 Atom)) (=>
      (in_2 y0 y1 (a2r_2 x0 x1))
      (and
         (= x0 y0)
         (= x1 y1))))))

; axiom for the unary operator "lone"
(forall ((X Rel1)) (=
   (lone_1 X)
   (forall ((a0 Atom)(b0 Atom)) (=>
      (and
         (in_1 a0 X)
         (in_1 b0 X))
         (= a0 b0)))))
```

6.4 Axioms for an ordered signature B

These are the basic axioms for an ordered signature B. The functions at, ord and $finite$ are generic; other functions are declared for B only.

```
;; functions
(declare-fun at (Rel1 Int) Atom)
(declare-fun finite (Rel1) Bool)
(declare-fun ord (Rel1 Atom) Int)
(declare-fun firstB () Rel1)
(declare-fun lastB () Rel1)
(declare-fun nextB () Rel2)
(declare-fun nextsB (Rel1) Rel1)

; axiom for firstB
(= firstB (a2r_1 (at B 1)))

; axiom for ord
(forall ((R Rel1)(a Atom)(b Atom)) (=>
   (and
      (in_1 a R)
      (in_1 b R)
      (= (ord R a) (ord R b)))
   (= a b)))

; axiom for nextB
(forall ((a Atom)(b Atom)) (=>
   (and
      (in_1 a B)
      (in_1 b B))
      (=
         (in_2 a b nextB)
         (= (ord B b) (+ (ord B a) 1)))))

; axiom for the function 'nexts' of B
(forall ((e Rel1)) (=>
   (subset_1 e B)
   (=
      (nextsB e)
      (join_1x2 e (transClos nextB)))))

; axiom for at (the reverse of ord)
(forall ((R Rel1)(a Atom)) (=>
   (in_1 a R)
   (= (at R (ord R a)) a)))

; infinite axiom for lastB
(=>
   (not (finite B))
   (= lastB none))
```

6.5 Proof listing for the lemma ‘step 45’

Try this proof online at http://rise4fun.com/Z3/vjCn

```
(set-logic AUFLIA)
(set-option :produce-unsat-cores true)
(set-option :macro-finder true)
;; sorts
(declare-sort Atom)
(declare-sort Rel1)
(declare-sort Rel2)
(declare-fun join_1x2 (Rel1 Rel2) Rel1)
(declare-fun subset_1 (Rel1 Rel1) Bool)

;; funs
(declare-fun in_1 (Atom Rel1) Bool)
(declare-fun in_2 (Atom Atom Rel2) Bool)

;; axioms
(assert (! ; axiom for join_1x2
  (forall ((A Rel1)(B Rel2)(y0 Atom)) (=
    (in_1 y0 (join_1x2 A B))
(exists ((x Atom)) (and (in_1 x A) (in_2 x y0 B)))))
 :named axiomc43ab575 ))
(assert (! ; subset axiom for Rel1
  (forall ((x Rel1)(y Rel1)) (=
    (subset_1 x y)
(forall ((a0 Atom)) (=> (in_1 a0 x) (in_1 a0 y)))))
 :named axiom76d2de83 ))

;; verification condition
; lemma about subsets within joins, from com-theorem1, related to step 45
(assert (!
  (not
    (forall ((a Rel1)(A Rel1)(R Rel2)) (=>
  (subset_1 a A)
  (subset_1 (join_1x2 a R) (join_1x2 A R)))))
 :named step45 )) ; :named lemma1aecfc94

(check-sat)
;(get-model)
(get-unsat-core)
```

Bibliography

[1] AlloyPE: an SMT-based proof engine for Alloy. http://asa.iti.kit.edu/305.php.

[2] Z3 - guide. http://rise4fun.com/Z3/tutorialcontent/guide, 2012.

[3] Konstantine Arkoudas, Sarfraz Khurshid, Darko Marinov, and Martin Rinard. Integrating model checking and theorem proving for relational reasoning. In *(RMICS)*, 2003.

[4] Kelloy. http://asa.iti.kit.edu/306.php.

[5] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[6] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[7] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[8] Paulson L. C Blanchette J., Böhme S. Extending sledgehammer with smt solvers. In *Automated Deduction - CADE-23*, volume 6803 of *Lecture Notes in Computer Science*, pages 116–130. Springer Berlin Heidelberg, 2011.

[9] Sascha Böhme. *Proving Theorems of Higher-Order Logic with SMT Solvers*. PhD thesis, Technische Universität München, 2012.

[10] Greg Nelson David Detlefs and James B. Saxe. Simplify: a theorem prover for program checking. J. ACM 52(3):365 - 473, 2005.

[11] Leonardo de Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In *Proceedings of the 21st international conference on Automated Deduction: Automated Deduction*, CADE-21, pages 183–198, Berlin, Heidelberg, 2007. Springer-Verlag.

[12] Aboubakr Achraf El Ghazi and Mana Taghdiri. Relational reasoning via SMT solving. In *17th International Symposium on Formal Methods (FM)*, pages 133–148, June 2011.

[13] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Dpll(t): Fast decision procedures. In *CAV*, pages 175–188, 2004.

[14] Ulrich Geilmann. *Verifying Alloy Models using KeY*. Diplomarbeit, Karlsruhe Institute of Technology, August 2011.

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

[16] Daniel Jackson and Kevin J. Sullivan. Com revisited: tool-assisted modelling of an architectural framework. In *SIGSOFT FSE*, pages 149–158, 2000.

[17] K. R. M. Leino and R. Monahan. Reasoning about comprehensions with first-order SMT. *(SAC)*, pages 615–622, 2009.

[18] M. Moskal, J. Łopuszański, and J. R. Kiniry. E-matching for fun and profit. In *Electronic Notes in Theoretical Computer Science 198*, volume 2, 2008.

[19] Mattias Ulbrich, Ulrich Geilmann, Aboubakr Achraf El Ghazi, and Mana Taghdiri. A proof assistant for Alloy specifications. In *18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 422–436, March 2012.

[20] Ulrich Geilmann. Verifying Alloy Models using KeY. Diplomarbeit, Karlsruhe Institute of Technology, August 2011.

[21] Jan van Eijck. Defining (reflexive) transitive closure on finite models. `http://homepages.cwi.nl/~jve/papers/08/pdfs/FinTransClosRev.pdf`.