Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 128 (2005) 37–52
www.elsevier.com/locate/entcs

Towards a Small Model Theorem for Data Independent Systems in Alloy

Lee Momtahan[1]

Oxford University Computing Laboratory, Oxford OX1 3QD, England

Abstract

Alloy is an extension of first-order logic for modelling software systems. Alloy has a fully automatic analyser which attempts to refute Alloy formulae by searching for counterexamples within a finite scope. However, failure to find a counterexample does not prove the formula correct. A system is data-independent in a type T if the only operations allowed on variables of type T are input, output, assignment and equality testing. This paper gives a theorem in a language closely related to Alloy, which applies to models of data-independent systems. The theorem calculates for such types T a threshold size. If no counterexamples are found at the threshold, the theorem guarantees that increasing the scope on T beyond the threshold still yields no counterexamples, and can complete the analysis for data-independent systems.

Keywords: Data independence, model finding, Alloy.

1 Introduction

Automated tools which can check the validity of logic formulae are important and versatile. Many practical problems can be reduced to the question of whether some formula is valid in some logical theory.

Model-finding is an alternative to the common approach of theorem proving. Model finders attempt to refute a formula by searching for a counterexample. This search is bounded by a user-determined finite *scope*, and whilst the existence of a counterexample shows the formula invalid, failure to find a counterexample does not prove the formula valid. The analysis is incomplete.

[1] Email: leemo@comlab.ox.ac.uk

doi:10.1016/j.entcs.2005.04.003

Alloy [3,4] is an extension of first-order logic for use in software modelling. The Alloy Analyser is the associated model-finder, and requires a scope to determine for each relevant type variable, the maximum size of the carrier set used in its search for counterexamples. The user must pick the size of carrier set for each type variable, when running the Alloy Analyser.

The contribution of this paper is to give a *small model theorem (SMT)*, which applies in the particular context of Alloy. The SMT produces for some formulae, a *threshold scope*. If the Alloy Analyser determines no counterexample exists within the threshold scope, the SMT proves that none exists at *any* scope, thereby completing the analysis. The SMT brings decidability results to a fragment of Alloy, by leveraging the Alloy Analyser.

In fact, the theorem does not usually produce an overall threshold scope for all type variables, but is still beneficial. The current theorem usually produces a threshold on a single type variable T, and once the user has determined the size of the other type variables, the theorem gives a threshold scope, with a guarantee that if no counterexamples are found at this scope, none exist by increasing the size of carrier set of T only. This relieves the user from determining some aspects of the scope, and may complete the check if the user can determine a priori bounds on their set of type variables. Future work will look at improving the theorem to deal with multiple type variables in a wider variety of cases.

Overview. Section 2 briefly introduces Alloy and looks at related work in data-independence and decidable fragments of first-order logic. Section 3 gives a formal definition of a slightly modified version of Alloy's kernel language. Then an example problem is presented in Section 4 and a threshold scope is determined for one of the base types in the problem using an informal argument. The following section gives a formal proof of the SMT. The final section discusses how a SMT might be generated for an unmodified Alloy kernel language, and other possible enhancements to the SMT.

2 Alloy and other related work.

Alloy. Alloy [3,4] is a modelling language which consists of first-order logic with sets and relations. It is roughly a subset of the Z notation [8], and also has similarities with UML's OCL [5,10]. Alloy is designed to bring to Z-style specification the kind of automation offered by model checkers.

The Alloy Analyser is a model finder for the Alloy language. The Analyser works by first translating formulae in the Alloy language into a smaller Alloy kernel language (AKL)[3]. Because the Analyser is given a finite scope, it can transform an AKL formula into a boolean formula, such that the boolean

formula has a model exactly when the Alloy kernel language formula (and hence the original formula) has a model within the given scope [2]. To test the boolean formula, the Analyser wraps off-the-shelf SAT solvers, such as SATO [12] or RelSAT [1].

Data-independence. Informally, a system is *data-independent* with respect to a type T if it can only input, output and store values of this type as well as copy them within its variables. The control-flow of such a system is not affected by different values; changing the input data will not affect behaviour except for the corresponding output data. Because the control-flow is independent of the type used this can be exploited in the verification of such systems.

These strict conditions on data-independence can often be relaxed to allow equality testing between variables of the type, and uninterpreted constants and finite range functions on the type as well, whilst still maintaining decidability results.

Data-independent systems are very common, for example a communication protocol is usually data-independent in the type communicated. Memory or database systems may be data-independent with respect to the type of values which they store as well as the type of address.

One can check data-independent systems through finite-instantiation methods. Threshold theorems can be developed which show that once a system is verified for all sizes of its data-independent type up to a particular value, then the system is correct for all non-empty instantiation of the type [11].

Decidable fragments of first-order logics. Decision procedures for fragments of (extensions of) first-order logic have been studied extensively by both mathematicians and in the context of formal verification. As mentioned in the introduction the SMT sometimes gives a threshold on every type variable relevant to a particular formula, which in conjunction with the Alloy Analyser, gives rise to a decision procedure for that formula. Although coinciding with known decidability results in such cases, the value of this work is to bring these results into the particular framework and setup of Alloy.

The more novel contribution of this work occurs when some of the type variables in a formula satisfy restrictions to allow the SMT to give thresholds, but other type variables are used without restriction. Once the user of the model finder has the set scope on the freely-used variables, the SMT can generate thresholds on the others. This does not give a decision procedure for the formula in question, and cannot be seen as coinciding with known decidability results, but of course still benefits the user.

3 Modified Alloy kernel language

After making some preliminary definitions, this section introduces a language based on the Alloy kernel language (AKL) [2], called the *Modified Alloy kernel language (AKL-M)*. This is the language to which the small model theorem in Section 5 applies. The AKL and AKL-M are very similar but the reasons for the differences and how they can be eliminated in future versions of the SMT are discussed in Section 6.

Images and preimages. Given a relation $r : \mathbb{P}(A \times B)$, for any $a_0 : A$, the *image of* a_0 *under* r, written $a_0 \langle\!\langle r$ is defined as[2] $\{(a, b) : r \mid a = a_0 \bullet b\}$. Similarly, for any $b_0 : B$, the *preimage of* b_0 *under* r, written $r \rangle\!\rangle b_0$ is defined as $\{(a, b) : r \mid b = b_0 \bullet a\}$.

Relational operators. If $r : \mathbb{P}(A \times B)$ then $r^{\sim}$ denotes the transpose of r. If $r : \mathbb{P}(A \times A)$ then r^{+} denotes the transitive closure of r. If $r : \mathbb{P}(A \times B)$ and $s : \mathbb{P}(B \times C)$ then $r \fatsemi s$ denotes the relational composition of r and s.

Multiplicity markings. Let $M = \{\mathtt{*}, \mathtt{+}, \mathtt{!}, \mathtt{?}\}$. The symbols $\mathtt{*}, \mathtt{+}, \mathtt{!}, \mathtt{?}$ are called *multiplicity markings* and are respectively used to denote: any, at least one, exactly one, up to one. For each $m \in M$ define a predicate σ_m as follows:

$$\sigma_{\mathtt{*}}(n) \Leftrightarrow \mathit{True}, \quad \sigma_{\mathtt{+}}(n) \Leftrightarrow n \geq 1, \quad \sigma_{\mathtt{!}}(n) \Leftrightarrow n = 1, \quad \sigma_{\mathtt{?}}(n) \Leftrightarrow n \leq 1$$

Type syntax. Let *TypeVar* denote a set of names for type variables and let $\mathtt{Unit} \in \mathit{TypeVar}$ be a distinguished name. The syntax of types is

$$\mathit{TypeExp} ::= \mathit{TypeVar}\ M\ \texttt{->}\ M\ \mathit{TypeVar}$$

For any type expression $Y\ m\ \texttt{->}\ n\ Z$, define $\mathit{Free}(Y\ m\ \texttt{->}\ n\ Z) = \{Y, Z\}$.

Atoms, set maps and carrier sets. Let *Atom* be a set whose elements are called *atoms*. A *set map* is a partial map from *TypeVar* to sets of atoms which: has a finite domain; is such that any two distinct elements of its domain are mapped to sets which are disjoint; and maps `Unit` to a singleton set. The image of a type variable under this map is called its *carrier set*.

Type expression semantics. For any type expression t and any set map δ such that $\mathit{Free}(t) \subseteq \operatorname{dom}\delta$ the *denotational semantics* of t with respect to δ, written $[\![t]\!]_\delta$, is defined as follows:

$$[\![Y\ m\ \texttt{->}\ n\ Z]\!]_\delta = \{r : \mathbb{P}(\delta(Y) \times \delta(Z)) \mid (\forall\, y : \delta(Y) \bullet \sigma_m(\#y \langle\!\langle r)) \\ \wedge\ (\forall\, z : \delta(Z) \bullet \sigma_n(\#r \rangle\!\rangle z))\}$$

[2] This notation is based on the Z standard [8]. A set comprehension $\{x : X \mid p(x) \bullet f(x)\}$ is formed by applying f to each x in X where $p(x)$ holds. See [9] sec. 5.2.

For example, Y `* -> *` Z means the relations between Y and Z, and Y `* -> !` Z denotes the total functions from Y to Z, etc. `Unit * -> !` Z represents an element of Z and `Unit * -> *` Z represents a subset of Z.

Expression and formula syntax. Let *Var* denote a set of names of variables. The syntax of expressions and formulae is defined as follows:

Expr ::=		*Formula* ::=	
Expr `+` *Expr*	*union*	*Expr* `in` *Expr*	*subset*
| *Expr* `&` *Expr*	*intersection*	| `!` *Formula*	*negation*
| *Expr* `-` *Expr*	*difference*	| *Formula* `&&` *Formula*	*conjunction*
| *Expr* `.` *Expr*	*navigation*	| *Formula* `||` *Formula*	*disjunction*
| `~` *Expr*	*transpose*	| `all` *Var* `:` *TypeVar* `|` *Formula*	*universal*
| `+` *Expr*	*closure*	| `some` *Var* `:` *TypeVar* `|` *Formula*	*existential*
| `{` *Var* `:` *TypeVar* `|` *Formula* `}`	*comprehension*		
| *Var*	*variable*		

Type maps and signatures. A *type map* is a partial map from *Var* to *TypeExp* with finite domain. A *signature* is a pair (Υ, Γ) where $\Upsilon \subseteq \mathit{TypeVar}$, Γ is a type map and $\forall\, v : \mathrm{dom}\,\Gamma \bullet \mathit{Free}(\Gamma(v)) \subseteq \Upsilon$.

Reduced type expressions. Although multiplicity markings are important to the semantics of a type expression, they are not used in type judgements. The following function is defined in order to strip multiplicity markings from a type expression: $\mathit{Strip}(Y\ m$ `->` $n\ Z) = Y$ `->` Z.

Type judgement. Given a type map, Γ, the type system of the language is defined by natural deduction as indicated in the table that follows. In the table: W, Y, Z denote type variables, v denotes a variable, e, d denote expressions, and F, G denote formulae. The type map $\Gamma, v : Y$ `->` Z stands for $\Gamma \oplus \{v \mapsto (Y$ `* -> *` $Z)\}$ i.e. the type map which is identical to Γ, but maps v to Y `* -> *` Z. The rules for d `&` e and d `-` e are like d `+` e. The rule for F `||` G is like F `&&` G. The rule for `some` v`:`Y`|`F is like `all` v`:`Y`|`F. For brevity they are omitted.

$$\frac{}{\Gamma \vdash v : t}\ [\ \mathit{Strip}(\Gamma(v)) = t\ ] \qquad \frac{\Gamma, v : \texttt{Unit -> } Y \vdash F}{\Gamma \vdash \{v : Y \mid F\} : \texttt{Unit -> } Y}$$

$$\frac{\Gamma \vdash d : Y \texttt{ -> } Z,\ \Gamma \vdash e : Y \texttt{ -> } Z}{\Gamma \vdash d \texttt{ + } e : Y \texttt{ -> } Z} \qquad \frac{\Gamma \vdash d : Y \texttt{ -> } Z,\ \Gamma \vdash e : Y \texttt{ -> } Z}{\Gamma \vdash d \texttt{ in } e}$$

$$\frac{\Gamma \vdash d : W \texttt{ -> } Y,\ \Gamma \vdash e : Y \texttt{ -> } Z}{\Gamma \vdash d \texttt{ . } e : W \texttt{ -> } Z} \qquad \frac{\Gamma \vdash F}{\Gamma \vdash \texttt{!}F}$$

$$\frac{\Gamma \vdash e : Y \texttt{ -> } Z}{\Gamma \vdash \texttt{~}e : Z \texttt{ -> } Y} \qquad \frac{\Gamma \vdash F,\ \Gamma \vdash G}{\Gamma \vdash F \texttt{ \&\& } G}$$

$$\frac{\Gamma \vdash e : Y \texttt{ -> } Y}{\Gamma \vdash \texttt{+}e : Y \texttt{ -> } Y} \qquad \frac{\Gamma, v : \texttt{Unit -> } Y \vdash F}{\Gamma \vdash \texttt{all } v : Y \mid F}$$

An expression e is defined to be *well-typed* with respect to Γ provided there exist $Y, Z \in \mathit{TypeVar}$ such that $\Gamma \vdash e : Y \texttt{ -> } Z$ can be derived. Similarly, a formula F is defined to be *well-typed* with respect to Γ provided $\Gamma \vdash F$.

Compatible signatures and formulae. A signature (Υ, Γ) and a formula F (resp. expression) are defined to be *compatible* provided F is well-typed with respect to Γ and any type variables appearing in F (i.e. introduced through quantification or set comprehension constructs) belong to Υ.

Instantiation. An *instantiation* of a signature (Υ, Γ) is an ordered pair (δ, η), where δ is a set map with domain Υ, and η is a total function with domain $\operatorname{dom}\Gamma$ such that: $\forall\, v : \operatorname{dom}\Gamma \bullet \eta(v) \in [\![\Gamma(v)]\!]_\delta$.

Language semantics. Given a signature and a formula F (resp. expression e) which are compatible, for any instantiation (δ, η) of the signature, the *denotational semantics* of F with respect to (δ, η), written $[\![F]\!]^\eta_\delta$ is defined by the table below. In the table, $\{\mathit{unit}\} = \delta(\texttt{Unit})$. (N.B. Expressions are thus interpreted as relations; formulae as booleans)

$$
\begin{array}{ll}
[\![d \texttt{ + } e]\!]^\eta_\delta = [\![d]\!]^\eta_\delta \cup [\![e]\!]^\eta_\delta & [\![v]\!]^\eta_\delta = \eta(v) \\
[\![d \texttt{ \& } e]\!]^\eta_\delta = [\![d]\!]^\eta_\delta \cap [\![e]\!]^\eta_\delta & [\![d \texttt{ in } e]\!]^\eta_\delta = [\![d]\!]^\eta_\delta \subseteq [\![e]\!]^\eta_\delta \\
[\![d \texttt{ - } e]\!]^\eta_\delta = [\![d]\!]^\eta_\delta \setminus [\![e]\!]^\eta_\delta & [\![\texttt{!}F]\!]^\eta_\delta = \neg\, [\![F]\!]^\eta_\delta \\
[\![d \texttt{ . } e]\!]^\eta_\delta = [\![d]\!]^\eta_\delta \fatsemi [\![e]\!]^\eta_\delta & [\![F \texttt{ \&\& } G]\!]^\eta_\delta = [\![F]\!]^\eta_\delta \wedge [\![G]\!]^\eta_\delta \\
[\![\texttt{\~{}}e]\!]^\eta_\delta = ([\![d]\!]^\eta_\delta)^\sim & [\![F \texttt{ || } G]\!]^\eta_\delta = [\![F]\!]^\eta_\delta \vee [\![G]\!]^\eta_\delta \\
[\![\texttt{+}e]\!]^\eta_\delta = ([\![d]\!]^\eta_\delta)^+ &
\end{array}
$$

$$
\begin{array}{l}
[\![\texttt{\{}v : Y \mid F\texttt{\}}]\!]^\eta_\delta = \{y : \delta(Y) \mid [\![F]\!]^{\eta \oplus \{v \mapsto \{(unit, y)\}\}}_\delta \bullet (unit, y)\} \\
[\![\texttt{all } v : Y \mid F]\!]^\eta_\delta = \forall\, y : Y \bullet [\![F]\!]^{\eta \oplus \{v \mapsto \{(unit, y)\}\}}_\delta \\
[\![\texttt{some } v : Y \mid F]\!]^\eta_\delta = \exists\, y : Y \bullet [\![F]\!]^{\eta \oplus \{v \mapsto \{(unit, y)\}\}}_\delta
\end{array}
$$

Consistent and valid formulae. Given a formula F and signature (Υ, Γ) which are compatible, F is defined to be *consistent* if and only if there exists an instantiation of (Υ, Γ), say (δ, η), such that $[\![F]\!]^\eta_\delta$. F is defined to be *valid* provided $\texttt{!}F$ is not consistent.

Scope. Given a signature (Υ, Γ), a *scope* Θ for (Υ, Γ) is defined to be a total function from Υ to $\mathbb{N} \cup \infty$. A scope Θ is defined to be *infinite* if and only if $\infty \in \operatorname{ran} \Theta$, otherwise it is *finite*.

Consistent and valid within scope. Given a formula F and signature (Υ, Γ) which are compatible and a scope Θ for (Υ, Γ), F is defined to be *consistent within* Θ if and only if there exists an instantiation of (Υ, Γ), say (δ, η), such that $[\![F]\!]^\eta_\delta$ and $\forall\, Y : \operatorname{dom} \delta \bullet \#(\delta(Y)) \leq \Theta(Y)$. F is defined to be *valid within* Θ if and only if $\texttt{!}F$ is not consistent within Θ.

4 Birthday book example

In this section Alloy is used to model a simple Birthday book program[3], and some assertions about the program are checked. An informal argument for threshold generation is presented.

```
sig Name {}
sig Date {}
sig BirthdayBook {known: set Name, birthday: known ->! Date}
fun AddBirthday (bb, bb': BirthdayBook, n: Name, d: Date) {
    bb'.birthday = bb.birthday ++ (n->d)
    }
fun DelBirthday (bb, bb': BirthdayBook, n: Name) {
    bb'.birthday = bb.birthday - (n->Date)
    }
fun FindBirthday (bb: BirthdayBook, n: Name, d: option Date) {
    d = bb.birthday[n]
    }
```

[3] The example originates in [8] and the translation to Alloy is given as an example in the Alloy distribution found at http://alloy.mit.edu/.

```
assert AddWorks {
    all bb, bb': BirthdayBook, n: Name, d: Date, d': option Date |
        AddBirthday (bb,bb',n,d) && FindBirthday (bb',n,d') => d = d'
    }
assert DelIsUndo {
    all bb1,bb2,bb3: BirthdayBook, n: Name, d: Date |
        AddBirthday (bb1,bb2,n,d) && DelBirthday (bb2,bb3,n)
            => bb1.birthday = bb3.birthday && bb1.known=bb2.known
    }
check AddWorks for 3 but 2 BirthdayBook
check DelIsUndo for 3 but 2 BirthdayBook
```

Thus `Name`, `Date`, `BirthdayBook`, are defined to be the type variables. For each element of `BirthdayBook`, `known` is a subset of `Name` and `birthday` is a total function from `known` to `Date`. (`->` signifies a general relation in this context, and `!` is a multiplicity marking, making this relation a total function.)

`AddBirthday` models the operation of adding an entry (the maplet `n->d`) to the birthday book. The override operator (`++`) is used to override any entry that may already be present for the name (`n`). `DelBirthday` models the operation of removing an entry for a specified name. The `FindBirthday` function models the operation of looking up a person's name to find their birthday.

`AddWorks` asserts that if a person's birthday is added to the birthday book, and then that person is looked up, the date that is returned is the date that was added. `DelIsUndo` asserts that if a person's birthday is added, and then that person is removed, the overall effect is to leave the birthday book unchanged from its original state.

The statement `check DelIsUndo for 3 but 2 BirthdayBook` gives the scope of the check of the `DelIsUndo` assertion: `Name` and `Date` have 3 elements, and `BirthdayBook`, 2. The following counterexample is obtained when this check is run in the Alloy Analyser:

```
BirthdayBook = {BirthdayBook_0, BirthdayBook_1}
Date = {Date_0, Date_1, Date_2}
Name = {Name_0, Name_1, Name_2}
known = {BirthdayBook_0 -> {},
         BirthdayBook_1 -> {Name_2} }
birthday = {BirthdayBook_0 -> {},
            BirthdayBook_1 -> {Name_2 -> Date_2} }
bb1@1 = BirthdayBook_1
bb2@2 = BirthdayBook_1
bb3@3 = BirthdayBook_0
n@4 = Name_2
d@5 = Date_2
```

This is interpreted as follows. Initially the birthday book contains a single entry: `Name_2` has their birthday on `Date_2`. Then, an entry is added, which

happens to be the same. Since it is the same the birthday book is unchanged. Then the entry for `Name_2` is deleted. This results in an empty birthday book. Thus the initial and final states of the birthday book are distinct, producing a counterexample.

The variables `bb1@1`, `bb2@2`, `bb3@3`, `n@4`, `d@5`, are skolem variables and represent the values the universally quantified variables of the formula take in the counterexample. Recall the Alloy Analyser works by trying to find a model of the negation of the formula whose validity is in question. After negating the formula, it is converted to negation normal form (NNF) and skolemized [2]. To convert to NNF, negations are pushed inside quantifiers using De Morgan's laws. Skolemization eliminates existentially quantified variables. If a variable is existentially quantified in a formula that is enclosed by no universal quantifiers, it can be replaced by a scalar. If a variable is existentially quantified in a formula that is enclosed by a universal quantifier, it is instead replaced by a function. Of course a formula has a model if and only if its skolemized NNF has one.

When the Alloy Analyser runs the `AddWorks` check (at the same scope) no counterexamples are found. This does not however prove the assertion valid and begs the question: could a counterexample be found at a larger scope? In particular, could a counterexample could be found by increasing the scope on `Date` only?

A threshold on `Date` is generated by the following argument. Firstly, the `AddWorks` assertion is negated, translated to NNF and skolemized. This produces skolem variables `bb@0`, `bb'@1`, `n@2`, `d@3` and `d'@4`. Then, suppose that this formula has a model when the scope on `Date` is infinite. Whatever the assignment of variables in the model, the assignment of variables involving `Date` are represented by a table where each row corresponds to a particular value of `Date`. For example, the following assignment of variables:

```
BirthdayBook = {BirthdayBook_0, BirthdayBook_1}
Date = {Date_0, Date_1, Date_2, ... }
Name = {Name_0, Name_1, Name_2}
known = {BirthdayBook_0 -> {Name_0, Name_1, Name_2},
         BirthdayBook_1 -> {Name_0, Name_1, Name_2} }
birthday = {BirthdayBook_0 ->
               {Name_0 -> Date_0, Name_1 -> Date_1, Name_2 -> Date_2},
            BirthdayBook_1 ->
               {Name_0 -> Date_0, Name_1 -> Date_1, Name_2 -> Date_3} }
bb@0 = BirthdayBook_0
bb'@1 = BirthdayBook_1
n@2 = Name_0
d@3 = Date_4
d'@4 = Date_4
```

would be represented in a table (with infinitely many rows) as:

	BirthdayBook_0			BirthdayBook_1				
	Name_0	Name_1	Name_2	Name_0	Name_1	Name_2	d@3	d'@4
Date_0	√			√				
Date_1		√			√			
Date_2			√					
Date_3						√		
Date_4							√	√
Date_5								
...								
Date_∞								

Then, this table is transformed to obtain a new table:

	BirthdayBook_0			BirthdayBook_1				
	Name_0	Name_1	Name_2	Name_0	Name_1	Name_2	d@3	d'@4
{Date_0}	√			√				
{Date_1}		√			√			
{Date_2}			√					
{Date_3}						√		
{Date_4}							√	√
{Date_5... Date_∞}								

This resulting table is formed by choosing values of `Date` which are the classes of the values of `Date` having equivalent rows in the preceding table. These actual values of `Date` (i.e. atoms), are not important to whether the assignment is a model; all that matters is that there are six of them and they are distinct. The above transformation can be repeated on any potential assignment. In the former table, '√' may occur no more than once in each column (due to multiplicity constraints), so there are no more than 8 rows, which do not contain ' ' (i.e. blank) throughout. Therefore there will never be more than 9 equivalence classes for any assignment, so a maximum of 9 rows in the latter table.

Because the formula in question does not use quantifier or set builder constructs over the `Date` type, the original assignment is a model if and only if the transformed assignment is a model (with the assignment of variables not covered by the table, i.e. not involving `Date`, unchanged). Thus 9 is a threshold for `Date`. The Alloy Analyser finds no models when the scope on `Date` is 9. (This includes a search for models when then scope is less than 9 due to a monotonicity property of the language.) One can be sure that increasing the scope on `Date` in the search for a model is fruitless.

If the scope on `BirthdayBook` or `Name` were to increase, the threshold on

`Date` would increase accordingly to $\#\texttt{BirthdayBook} \times \#\texttt{Name} + 3$. An overall threshold scope has not been generated, but this may be possible in future work.

5 Small model theorem

In this section, a SMT about the AKL-M (Section 3) is derived. The gross structure of the proof is given, but in the interests of brevity, derivations of lemmas are only outlined. Full derivations can be found in [7].

Distinguished type variable. Without loss of generality it is assumed that the type variable for which the SMT generates a threshold is $\boldsymbol{X}$. This is a fixed entity throughout the proof and therefore a bold type face is used. Various restrictions on the use of $\boldsymbol{X}$ are developed throughout the proof and summarized at the end of this section.

Equivalent atoms. In the context of a signature (Υ, Γ) and a set map δ, a family of equivalence relations on $\delta(\boldsymbol{X})$ is defined below. The relations are indexed by type expressions t and possible meanings of those type expressions $val : [\![t]\!]_\delta$ and written as an infix operator: $\sim(val, t)$.

$$x \sim(val,\ Y\ m \texttt{->} n\ Z)\ x' \Leftrightarrow ((Y = \boldsymbol{X}) \Rightarrow x \langle\!\langle val = x' \langle\!\langle val) \wedge ((Z = \boldsymbol{X}) \Rightarrow val \rangle\!\rangle x = val \rangle\!\rangle x')$$

In the context of a signature (Υ, Γ) and an instantiation of it (δ, η), a further equivalence relation on $\delta(\boldsymbol{X})$, written $\sim$ is defined by:

$$x \sim x' \Leftrightarrow \forall (v \mapsto t) : \Gamma \bullet x \sim(\eta(v), t)\ x'$$

For notational convenience, the definition of $\sim$ is extended to every carrier set. On carrier sets of type variables other than $\boldsymbol{X}$, $\sim$ is the identity relation.

Equivalence Classes. For any atom y let $[y]^\sim$ denote the equivalence class of y. For any type variable Y, let $[]^\sim_Y$ denote the set $\{y : \delta(Y) \bullet [y]^\sim\}$.

Quotient of an instantiation. In the context of a given signature $S = (\Upsilon, \Gamma)$ and instantiation of it $I = (\delta, \eta)$ the *quotient* of I is denoted $\overline{I} = (\overline{\delta}, \overline{\eta})$, and defined by:

$$\begin{aligned}
&\operatorname{dom} \overline{\delta} = \Upsilon \\
&\forall\, Y : \Upsilon \bullet \overline{\delta}(Y) = []^\sim_Y \\
&\operatorname{dom} \overline{\eta} = \operatorname{dom} \eta \\
&\forall\, v : \operatorname{dom} \overline{\eta} \bullet \overline{\eta}(v) = \{(y, z) : \eta(v) \bullet ([y]^\sim, [z]^\sim)\}
\end{aligned}$$

Quotient set maps and quotient instantiations. Let $\overline{Atom} = \mathbb{P}\, Atom$. A *quotient set map* is defined exactly like a set map (Section 3), except with $\overline{Atom}$ substituted for $Atom$. Similarly, a *quotient instantiation* is defined exactly like an instantiation (Section 3), except with quotient set map substituted for set map. For brevity these definitions are not repeated.

Lemma 5.1 *Let S be a signature and I be an instantiation of it. Then $\overline{I}$ is a quotient instantiation of S.*

Proof Outline: To prove that $\overline{I}$ is an instantiation, it is necessary to check $\forall\, v : \mathrm{dom}\,\Gamma \bullet \overline{\eta}(v) \in [\![\Gamma(v)]\!]_{\overline{\delta}}$. □

Functions relating an instantiation and its quotient. As will be shown, the semantics of an expression with respect to an instantiation and its quotient instantiation can be related using the following functions. Let $S = (\Upsilon, \Gamma)$ be a signature and $I = (\delta, \eta)$ be an instantiation of it. Let $Y\ m \mathbin{\texttt{->}} n\ Z$ be a type expression. Define:

$$K^I_S(Y \mathbin{\texttt{->}} Z) : [\![Y * \mathbin{\texttt{->}} * Z]\!]_\delta \to [\![Y * \mathbin{\texttt{->}} * Z]\!]_{\overline{\delta}}$$
$$L^I_S(Y \mathbin{\texttt{->}} Z) : [\![Y * \mathbin{\texttt{->}} * Z]\!]_{\overline{\delta}} \to [\![Y * \mathbin{\texttt{->}} * Z]\!]_\delta$$
$$K^I_S(Y \mathbin{\texttt{->}} Z)(r) = \{(y, z) : r \bullet ([y]^\sim, [z]^\sim)\}$$
$$L^I_S(Y \mathbin{\texttt{->}} Z)(r) = \bigcup\{(y, z) : r \bullet a \times b\}$$

Note that $\forall\, r : [\![Y * \mathbin{\texttt{->}} * Z]\!]_{\overline{\delta}} \bullet K^I_S(Y \mathbin{\texttt{->}} Z)(L^I_S(Y \mathbin{\texttt{->}} Z)(r)) = r$.

Banned constructs. A restricted set of formulae can be related with the above functions. The *banned constructs* are the following where $v : Var$:

$$\texttt{all}\ v\texttt{:}\boldsymbol{X}\texttt{|}\ldots, \quad \texttt{some}\ v\texttt{:}\boldsymbol{X}\texttt{|}\ldots, \quad \texttt{\{}v\texttt{:}\boldsymbol{X}\texttt{|}\ldots\texttt{\}}$$

i.e. Quantification or set comprehension over $\boldsymbol{X}$. However, set comprehension is not a banned construct in the following special case[4]: $\texttt{\{}v\texttt{:}\boldsymbol{X}\texttt{|}\ v\ \texttt{in}\ v\texttt{\}}$.

The next lemma applies only to formulae which do not use banned constructs.

Lemma 5.2 *Let e be an expression which does not use banned constructs. Let F be a formula which does not use banned constructs. Let $S = (\Upsilon, \Gamma)$ be*

[4] The syntax of the full Alloy language allows type variables to be used in expressions, and they are translated to the AKL using this construct. It is therefore desirable that the SMT accommodates it.

a compatible signature, and $I = (\delta, \eta)$ *be an instance of it. Then:*

$$(\Gamma \vdash e : Y \texttt{ -> } Z) \Rightarrow L^I_S(Y \texttt{ -> } Z)([\![e]\!]^{\overline{\eta}}_{\overline{\delta}}) = [\![e]\!]^{\eta}_{\delta}$$
$$(\Gamma \vdash F) \Rightarrow ([\![F]\!]^{\overline{\eta}}_{\overline{\delta}} \Leftrightarrow [\![F]\!]^{\eta}_{\delta})$$

Proof Outline: The proof of this lemma uses structural induction over the expression or formula. The base cases of expressions which are variables or use set comprehension in the particular mode which is not banned, follow from the definition of the quotient of an instantiation. □

Size. The following function will be used to generate a bound on the number of equivalence classes which partition the carrier set of the type variable $\boldsymbol{X}$. Its first argument is a signature, its second a scope.

$$Size(S, \Theta) = \infty \qquad \text{if } \exists (v \mapsto (Y\ m \texttt{ -> } n\ Z)) : \Gamma \bullet Y = \boldsymbol{X} \wedge Z = \boldsymbol{X}$$
$$Size(S, \Theta) = \textstyle\sum_{(v \mapsto t) \in \Gamma} Sum(t) + \prod_{(v \mapsto t) \in \Gamma} Prod(t) \qquad \text{otherwise}$$

where $Sum(t)$ is defined to be 0, except for any $Y : TypeVar \setminus \{\boldsymbol{X}\}$ and any multiplicity marking $m : \{\texttt{*}, \texttt{+}, \texttt{!}, \texttt{?}\}$:

$$Sum(Y\ m \texttt{ -> ! } \boldsymbol{X}) = \Theta(Y), \quad Sum(\boldsymbol{X} \texttt{ ! -> } m\ Y) = \Theta(Y)$$
$$Sum(Y\ m \texttt{ -> ? } \boldsymbol{X}) = \Theta(Y), \quad Sum(\boldsymbol{X} \texttt{ ? -> } m\ Y) = \Theta(Y)$$

and $Prod(t)$ is defined to be 1, except for any $Y : TypeVar \setminus \{\boldsymbol{X}\}$ and any multiplicity marking $m : \{\texttt{*}, \texttt{+}\}$:

$$Prod(Y \texttt{ ! -> } m\ \boldsymbol{X}) = \Theta(Y), \qquad Prod(\boldsymbol{X}\ m \texttt{ -> ! } Y) = \Theta(Y)$$
$$Prod(Y \texttt{ ? -> } m\ \boldsymbol{X}) = \Theta(Y) + 1, \quad Prod(\boldsymbol{X}\ m \texttt{ -> ? } Y) = \Theta(Y) + 1$$
$$Prod(Y \texttt{ * -> } m\ \boldsymbol{X}) = 2^{\Theta(Y)}, \qquad Prod(\boldsymbol{X}\ m \texttt{ -> * } Y) = 2^{\Theta(Y)}$$
$$Prod(Y \texttt{ + -> } m\ \boldsymbol{X}) = 2^{\Theta(Y)} - 1, \quad Prod(\boldsymbol{X}\ m \texttt{ -> + } Y) = 2^{\Theta(Y)} - 1$$

N.B. Although *Size* depends on Θ, is it independent of $\Theta(\boldsymbol{X})$.

Lemma 5.3 *Let* S *be a signature and* $I = (\delta, \eta)$ *be an instantiation of it. Then* $\#[]^{\sim}_{\boldsymbol{X}} \leq Size(S, \# \circ \delta)$.

Proof Outline: Referring back to the tabular construction in Section 4 one can see that if the table has a finite number of columns, then there are a finite number of equivalence classes, certainly no more than 2 to the power of the number of columns. Tighter bounds are obtained using the multiplicity marking information. □

Theorem 5.4 (Small model theorem) *Let F be a formula not using banned constructs and let S be a compatible signature. Let Θ be a scope for S. Let $\Theta' = \Theta \oplus \{\boldsymbol{X} \mapsto \mathit{Size}(S, \Theta)\}$. If F is valid within Θ' then F is valid within Θ.*

Proof It is sufficient to prove the equivalent statement: if $!F$ is consistent within Θ then $!F$ is consistent within Θ'. So suppose $!F$ is consistent within Θ. Then choose an instantiation $I = (\delta, \eta)$ such that $\forall\, Y : \mathrm{dom}\, \delta \bullet \#(\delta(Y)) \leq \Theta(Y)$ and $[\![!F]\!]^{\eta}_{\delta}$.

It follows that $\#\overline{\delta}(\boldsymbol{X}) = \#[]^{\sim}_{\boldsymbol{X}} \leq \mathit{Size}(S, \# \circ \delta)$ by the definition of the quotient of an instantiation and Lemma 5.3. But $\# \circ \delta \leq \Theta$ under the point-wise ordering and it follows from its definition that the function *Size* is monotone in its second argument. Thus $\mathit{Size}(S, \# \circ \delta) \leq \mathit{Size}(S, \Theta)$ and hence $\#\overline{\delta}(\boldsymbol{X}) \leq \mathit{Size}(S, \Theta) = \Theta'(\boldsymbol{X})$. Furthermore, since the notion of equivalence between atoms was extended to be just the identity relation for carrier sets other than $\boldsymbol{X}$'s it follows that: $\forall\, Y : \mathrm{dom}\, \overline{\delta} \setminus \{\boldsymbol{X}\} \bullet \#\overline{\delta}(Y) = \#\delta(Y)$. This establishes $\forall\, Y : \mathrm{dom}\, \overline{\delta} \bullet \#(\overline{\delta}(Y)) \leq \Theta'(Y)$.

But $[\![!F]\!]^{\eta}_{\delta}$, so by Lemma 5.2 it follows that $[\![!F]\!]^{\overline{\eta}}_{\overline{\delta}}$. Thus $!F$ is consistent within Θ'. □

Recap of conditions on $\boldsymbol{X}$. The conditions to obtain a threshold on $\boldsymbol{X}$ in the check of a formula are as follows. The compatible signature must not use any variable of type $\boldsymbol{X}$ m -> n $\boldsymbol{X}$. Also, the formula may not use quantification or set comprehension constructs over $\boldsymbol{X}$ (except in the special mode allowed - see: banned constructs). However, after translation to negation normal form, existential quantification over $\boldsymbol{X}$ can be effectively allowed by skolemizing [5] the formula before applying the above theorem.

6 Further work

In future it is hoped to implement automatic threshold generation in the Alloy Analyser by using the results of the SMT. However the current SMT applies to the Modified Alloy kernel language (AKL-M) not the true Alloy kernel language (AKL). The differences between AKL and the AKL-M concern the syntax of types. The syntax of types in AKL is:

$$\mathit{TypeExp}_{AKL} = \mathit{TypeVar} \texttt{ -> } \mathit{TypeVar} \mid \mathit{TypeVar} \texttt{ => } \mathit{TypeExp}_{AKL}$$

An AKL type expression is either a relation between type variables or a total function from a type variable to a type expression. This is the only difference

[5] In fact existential quantification over $\boldsymbol{X}$ nested inside more than one level of universal quantification (over types variables other than $\boldsymbol{X}$) can not be allowed in this language, but the richer type syntax proposed for future work would allow such nesting.

between AKL and AKL-M except for the corresponding function application construct of the language syntax.

The use of a recursively defined type syntax adds complexity to the proof of the SMT. The key to a SMT with such a type syntax seems to be *logical relations*, as used in [6].

Whereas AKL-M has multiplicity markings, but AKL does not, in some sense the SMT presented here is more general than required. The Alloy Analyser replaces multiplicity markings which are present in the full Alloy language formula with conjuncts to the formula giving the appropriate restrictions (see [3]). But the loss of multiplicity marking information in type expressions, yields much looser thresholds, possibly leading to state explosion. This difficulty may be overcome by compiling full Alloy language problems into a language based on AKL but with multiplicity markings (reusing most of the existing compiler). Then the SMT can be applied to generate thresholds.

A further piece of work is to improve the ability of the small model theorem to deal with multiple type variables. Currently the theorem can be applied repeatedly to multiple type variables, only if there are no relations between them. It may be possible to deal with multiple type variables when additional restrictions are imposed.

Acknowledgement

The author would like to thank Andrew Martin, Jeremy Gibbons, Ranko Lazić, Gavin Lowe and Bill Roscoe for their useful comments, and the UK EPSRC for its support.

References

[1] Roberto J. Jr. Bayardo and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208, Providence, Rhode Island, 1997.

[2] Daniel Jackson. Automating first-order relational logic. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 130–139. ACM Press, 2000.

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

[4] Daniel Jackson. Micromodels of software: Lightweight modelling and analysis with alloy. Technical report, MIT Lab for Computer Science, 2002.

[5] Ivar Jacobson Jame Rumbaugh and Grady Booch. *The Unified Modelling Langauge Reference Manual*. Addison-Wesley, 1999.

[6] Ranko Lazić and David Nowak. A Unifying Approach to Data-independence. In *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR 2000)*, volume 1877 of *Lecture Notes in Computer Science*, pages 581–595. Springer-Verlag, August 2000.

[7] Lee Momtahan. A simple small model theorem for Alloy. Technical report, Oxford University Computing Laboratory, June 2004.

[8] J. M. Spivey. *The Z notation : a reference manual*. Prentice-Hall, 1992.

[9] Jim Woodcock and Jim Davies. *Uzing Z: Specificiation, Refinement, and Proof*. Prentice Hall, 1996.

[10] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.

[11] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 184–193. ACM Press, 1986.

[12] Hantao Zhang. SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI*, pages 272–275, 1997.