Enhancing the Alloy Analyzer with Patterns of Analysis

William Heaven and Alessandra Russo

Department of Computing, Imperial College London, SW7 2AZ
{william.heaven, ar3}@doc.ic.ac.uk

Abstract. Formal techniques have been shown to be useful in the development of correct software. But the level of expertise required of practitioners of these techniques prohibits their widespread adoption. Formal techniques need to be tailored to the commercial software developer. Alloy is a lightweight specification language supported by the Alloy Analyzer (AA), a tool based on off-the-shelf SAT technology. The tool allows a user to check interactively whether given properties are consistent or valid with respect to a high-level specification, providing an environment in which the correctness of such a specification may be established. However, Alloy is not particularly suited to expressing program specifications and the feedback provided by AA can be misleading where the specification under analysis or the property being checked contains inconsistencies. In this paper, we address these two shortcomings. Firstly, we present a lightweight language called *Loy*, tailored to the specification of object-oriented programs. An encoding of Loy into Alloy is provided so that AA can be used for automated analysis of Loy program specifications. Secondly, we present some *patterns of analysis* that guide a developer through the analysis of a Loy specification in order to establish its correctness before implementation.

1 Introduction

Since the late 60s and the presentation of a logic for programs [7,8], a large field of Computer Science research has been concerned with the application of formal techniques to software development in order to provide some guarantee that the software will behave as expected [1]. Commercial pressure to produce higher-quality software is always increasing [6,4]. But despite the considerable advances made in this area from a research point of view, very few of these techniques are applied today in industry. On the whole, a combination of the level of expertise required of practitioners of these techniques and the apparent extra costs they impose on software development have made their adoption commercially prohibitive. Software developers favour so-called *lightweight* techniques that provide more immediate returns and that sit more comfortably with the activity of implementation [11].

Several lightweight specification languages supported by an array of tools have been proposed. For example, JML [15] and Spec♯ [2] allow implementers to annotate source files with formal specifications. These specifications can be used

by tools to check the correctness of an implementation, statically or at runtime [3,2]. Such tools are particularly suited to catching null-pointer and index-out-of-bounds exceptions or violations of invariants and method specifications in an implementation. However, they do not support a direct check of the consistency of a specification and inconsistency sometimes becomes manifest only through the unexpected results of analysis. For example, ESC/Java2, a tool that statically checks a Java implementation against a JML specification, will always pass a method body when checking it against its specification if the precondition of the method is unsatisfiable, since the tool does not check the bodies of methods that do not satisfy their precondition. To avoid misleading results and, of course, since an inconsistent specification has no possible implementation, a developer should be able to establish the consistency of a specification before implementation is attempted.

Alloy [9], another lightweight specification language, is supported by the Alloy Analyzer (AA) [10], a tool based on off-the-shelf SAT technology. The tool allows a user to check whether given properties are consistent or valid with respect to a specification, providing an environment in which the correctness of a specification may be established. However, the environment has two shortcomings for the software developer: Alloy is not specifically suited to expressing *program* specifications and feedback from AA can be misleading when the specification under analysis or property being checked contains inconsistencies.

Firstly, Alloy does not have an implicit concept of state. For example, a relation between the before and after value of a field f in a JML method specification can be specified in terms of a relation between the variables $\backslash old(f)$ and f. But Alloy has no comparable implicit means to denote the before and after value of a field. In Alloy, this relation would have to be expressed by explicitly constructing the necessary constraints between two intrinsically unrelated variables f and f', say. In practice, specifying these extra constraints is a complicated task and can lead to an overly cluttered specification.

Secondly, feedback from AA can be misleading when the specification under analysis or the property being checked contain inconsistencies. If a user of the tool is unaware of an existing inconsistency, accepting such feedback at face value can lead to further error. For example, consider the small Alloy specification shown below.

```
sig Project { }
sig Employee { project : Project }
sig Pool extends Employee { } { no project }
fact { some Pool }
```

The specification represents a scenario in which employees are assigned projects and some employees belong to a pool. Employees belonging to the pool have no assigned project. However, there is an inconsistency. The Alloy declaration *project : Project* indicates that each employee has a project but employees belonging to the pool are specified to have no project. Since there must be at least one employee in the pool (because of the constraint imposed by the *fact*

paragraph) the specification is unsatisfiable. Now, if AA were asked whether or not some property were consistent with the specification, such as whether there could be some employees not in the pool —

```
pred PropertyTest () { some e : Employee | e not in Pool }
```

— the tool would be unable to instantiate a model of the specification that satisfies this property because it is unable to instantiate a model of the specification itself. But AA simply suggests that the property may be inconsistent. If a user has no reason to suspect the fault of the specification, not realising that *nothing* is consistent with it, the tested property will be rejected. In short, simply informing a user that a model cannot be found, without warning of possible mis-specification, is not adequate feedback.

This paper addresses these two shortcomings. We present a lightweight language called *Loy*, tailored to the specification of object-oriented programs. In particular, Loy allows the specification of before and after state values of fields in method specifications. We implement analysis support for this language by encoding Loy specifications into Alloy, invoking AA on the encodings and feeding back results in terms of the original Loy specification. But, most importantly, we show how the checking of specifications written in this language can be supported by *patterns of analysis* that guide specifiers in questioning feedback from AA. Section 2 gives an overview of how AA checks an Alloy specification. Section 3 introduces the language Loy. Section 4 presents patterns of analysis for Loy specifications and Section 5 discusses an encoding of Loy into Alloy that allows automation of the patterns to be implemented on top of AA. Section 6 discusses related work and Section 7 concludes.

2 The Alloy Analyzer

An Alloy specification consists of a set of *signature paragraphs*, a set of *fact paragraphs*, a set of *predicate paragraphs*, a set of *assert paragraphs* and a set of *function paragraphs*. A signature paragraph declares a type and contains a set of field declarations for instances of that type. Semantically, a type is treated as a set, or *domain*, and its instances as the elements of that domain. A field is treated as a relation between domains. The constraints introduced implicitly by the signature declarations together with the constraints enclosed by the fact paragraphs constitute a set of *core constraints*, which are expected to be satisfied by all models of the specification. Predicate paragraphs enclose a set of formulae that can be tested for *consistency* with respect to a specification and assert paragraphs enclose a set of formulae that can be tested for *validity* with respect to a specification. Function paragraphs are essentially macro-expanded expressions and not relevant to the considerations of satisfiability in this paper. They are therefore not discussed further.

A *binding* is a function that maps variables of the specification according to their types to elements of the domains declared by the signature paragraphs. The semantics of a specification can be given by a set $\{C, P_1, \ldots, P_n, A_1, \ldots, A_m\}$ where C is a set of bindings associated with the core constraints, P_i is a set of

bindings associated with the constraints of a predicate paragraph i and A_j is a set of bindings associated with the constraints of an assert paragraph j. For a set of constraints c, $\vartheta \vDash c$ means that binding ϑ *satisfies* the constraints of c.

Definition 1 (Bindings for the core constraints) *Let γ be the set of core constraints. $C = \{\vartheta \mid \vartheta \vDash \gamma\}$ is the set of bindings associated with the constraints of γ. The bindings of C bind variables of the specification such that the constraints of γ are satisfied.*

The constraints of a predicate paragraph are expected to be consistent with respect to the core constraints. The bindings of P_i are those bindings that satisfy the core constraints and the constraints of predicate paragraph i of the specification.

Definition 2 (Bindings for a predicate paragraph) *Let γ be the set of core constraints and let i be a predicate paragraph. $P_i = \{\vartheta \mid \vartheta \vDash \gamma \wedge i\}$ is a set of bindings that bind variables of the specification and the variables of i (including the parameters of i) such that the constraints of γ and i are satisfied.*

The constraints of an assert paragraph are expected to be valid with respect to the core constraints. That is, there should be no binding that satisfies the core constraints but not the constraints of an assert paragraph. The bindings of A_j are those bindings that satisfy the core constraints but not the constraints of assert paragraph j of the specification. A_j is expected to be the empty set.

Definition 3 (Bindings for an assert paragraph) *Let γ be the set of core constraints and let j be an assert paragraph. $A_j = \{\vartheta \mid \vartheta \vDash \gamma \wedge \neg j\}$ is a set of bindings that bind variables of the specification and the variables of j such that the constraints of γ and $\neg j$ are satisfied.*

In practice, AA searches for a model of a specification within a given *scope*. That is, given a scope of 3, the tool searches for a model that satisfies a set of constraints such that there are at most three instances of each type in the model. Therefore, when the tool fails to find a model it may be because the specification is not satisfiable within the given scope, not that it is inconsistent in itself. However, the scope problem is orthogonal to the shortcoming addressed in this paper and it is assumed in this paper that AA never fails to find a model because of an insufficiently large scope.

3 Loy

Loy is a lightweight formal specification language for object-oriented programs. Its syntax is text-based and borrows keywords from common object-oriented program specification languages and its semantics is based on common notions of invariant and method specification [15]. Loy supports the specification of class interfaces within a single-inheritance hierarchy and includes the notions of subclass and subtype. Variables declared in a class specification are abstract specification variables [5] and methods are specified with preconditions, postconditions and frame conditions in a *design-by-contract* manner [19].

Loy also includes *depends clauses* to allow sound modular specification of the frame conditions [21]. The syntax of a Loy specification is given in Figure 1 (c, v, m and ϕ respectively denote a class name, a variable name, a method name and set of first-order formulae; *token*+ denotes one or more instances of *token* and *[token]* denotes zero or one instances of *token*). The main constructs of Loy are introduced below through three example class specifications.

‹spec›	::= ‹class›+	specification
‹cspec›	::= `class` c_1 `ext` c_2 ‹body› ‹mspec›+	class specification
‹body›	::= ‹dec›+ ‹dep›+ ‹inv›+	class body
‹dec›	::= v `:` c	field declaration
‹dep›	::= `depends` v_1 `<-` $v_2 \ldots v_n$	depends clause
‹inv›	::= `invariant` ϕ_I	invariant
‹mspec›	::= *[* c *]* m ‹dec›+ *[* ‹pre› *][* ‹post› *][* ‹mod› *]*	method specification
‹pre›	::= `requires` ϕ_P +	precondition
‹post›	::= `ensures` ϕ_Q +	postcondition
‹mod›	::= `modifies` $v_1 \ldots v_n$	frame condition

Fig. 1. Abstract syntax for a Loy specification

Example 1. (Project.loy)

```
class Project {
    manager : Manager
    invariant some manager
}
```

Example 2. (Employee.loy)

```
class Employee {
    project : Project
    invariant no project.manager

    assign (p : Project)
        requires no project
        ensures project' = p
        modifies project
}
```

Example 3. (ManagedEmployee.loy)

```
class ManagedEmployee ext Employee {
    manager : Manager
    depends manager <- project

    assign (p : Project)
        requires no project
        ensures project' = p
        ensures manager' = p.manager
        modifies project
}
```

The above class specifications describe aspects of a relationship between employees, projects and managers (the specification for the class *Manager* is not shown). A class *Project* (Example 1) has a field *manager* of type *Manager* that represents the manager of the project and an invariant that specifies that *manager* must not be null (*some manager*). The declarations and invariants constitute the core constraints of a Loy specification. Formulae are written in a first-order logic that includes the logical connectives *and*, *or*, *implies*, *not*, the quantifiers *all* and *exists* and the Alloy keywords *no* and *some* to specify that a set is empty and nonempty, respectively. A class *Employee* (Example 2) has a field *project* of type *Project* and an invariant that states that the em-

ployee's project has no manager (*no project.manager)*. A method *assign*, which takes a parameter of type *Project*, is specified: an employee can be assigned to a project only if no project has been assigned already. This constraint (given by the *requires* clauses) is the precondition of the method. The postcondition of the method (given by the *ensures* clause) simply states that the project p is assigned to the field *project* in the after state of the method. A field reference with a prime (e.g. *project* $'$) denotes the value of the field in the after state of a method. Field references appearing in method specifications without a prime denote values in the before state. The frame condition of the method (given by the *modifies* clause) states that only the value of the field *project* may be affected by an invocation of *assign*.

Employee is extended by a class *ManagedEmployee* (Example 3), which adds a field *manager* of type *Manager* to the field *project* inherited from *Employee*. However, the specification for *assign* is not inherited but overridden. The postcondition of the method is strengthened and the precondition and frame condition are unchanged. The postcondition now states that project p is assigned to *project* and the manager *p.manager* is assigned to *manager* in the after state of the method. The depends clause states that when the field *project* changes value, the field *manager* may also change value. Thus, the overriding version of *assign* may change the value of *manager* and yet have an equivalent frame condition to the overridden method, as the behavioural subtype relation requires [17].

Loy borrows features from both Alloy and JML. From Alloy it takes its first-order relational logic and from JML it takes its modular object-oriented specification structure. But, unlike Alloy, the concept of state in a Loy specification is semantically implicit and the syntactic constructs of the language are tailored to the specification of object-oriented programs. For example, we can simply denote the before and after state value of the field *manager* by writing *manager* and *manager* $'$. Further, mathematical constructs implicit in the declarations of Alloy fields do not appear in Loy, which requires all constraints to be expressed explicitly as invariants. This allows the consistency of these constraints to be more easily checked against the trivially satisfiable field declarations and depends clauses. For example, a field f in Alloy might be declared as f *: F* or f *: lone F*, indicating that f is nonempty or possibly empty, respectively. We can express that a field f is nonempty in Loy with the invariant *some f*. Similarly, while Loy allows only declarations of the form f *: F* and f *: set F*, Alloy allows binary relation types (f *: F -> G*) and set expression types (f *: F + G*, for union; f *: F - G*, for difference; and f *: F & G*, for intersection). In Loy, such constructs may be represented explicitly with user-defined data types. Finally, unlike JML, Loy specifications do not declare program variables. All variables of a Loy specification are abstract and no means of relating abstract variables to program variables is provided (c.f. *represents* clauses in JML [15] and Spec♯ [2]). This avoids the sometimes confusing distinction between abstract and program variables in a specification and the need to declare program variables with one access modifier for the implementation and another for the specification (c.f. the

spec_public access modifier in JML [15]).

Loy specifications can be analysed using AA through an encoding of Loy into Alloy, presented in Section 5. For example, the satisfiability of the specification of the method *assign* in *Employee* can be checked using AA by checking whether the Alloy encoding of the conjunction of the precondition, postcondition and frame condition is consistent with respect to the core constraints. The tool searches for a model of the specification that satisfies this conjunction and, failing to find one, suggests that the specification of the method may be inconsistent. What is wrong with the specification of *assign*? Of course, nothing is wrong with the specification of the method: the inconsistency lies in the specification of the invariants of the classes *Project* and *Employee*. The invariant for *Project* states that there must be a manager while the invariant for *Employee* states that its project has no manager. No model that satisfies the above property is found because there is no model that satisfies the specification of *Project* and *Employee*. But AA does not point a user in this direction.

4 Patterns of Analysis

This Section presents some *patterns of analysis* — one for each of the logical connectives and quantifiers of Loy — that may be applied to Loy specifications and automated with AA to address the shortcoming illustrated above. The patterns are presented in Figures 2–7 as decision trees in which a non-terminal node represents a satisfiability query submitted to AA with respect to a given specification. Note that it is only necessary to consider satisfiability queries because validity of a formula A is handled by checking the satisfiability of $\neg A$. If there is no model of the specification that satisfies $\neg A$ the tool informs the user that A is possibly valid.

It is assumed that the consistency of the core constraints of a specification is established before checking other properties, such as the correctness of method specifications. The consistency of the core constraints of a specification are therefore checked first by checking their consistency with $\top$. If the core constraints of a specification are inconsistent, further investigation is carried out by applying the patterns to each invariant in turn, checking its consistency with respect to the declarations of the specification and in conjunction with the other invariants.

Given a specification S, $\mathrm{SAT}(A)_T$ represents a query that asks whether there is a model of the core constraints of S that satisfies A, where any free variables in A are bound according to the type information held in the set T. At the start of an application of the patterns, T is empty but the patterns for universal and existential quantification (Figures 6 and 7) strip off the outer quantifier leaving the bound variable free. For example, if the pattern for universal quantification is applied to the formula $\forall\, x \in X \cdot A(x)$, the quantification $\forall\, x \in X$ is stripped off before applying the pattern for $A(x)$ and the information that x is of type X is recorded in T. It is also assumed that the declarations of a specification provide contextual information for the specification variables in A. For example, checking whether the precondition of *assign* (i.e. *no projects*) is satisfiable, involves checking whether there is an instance of *Employee*, e say, such that *no*

e.projects is satisfied. The information that *projects* is a field of *Employee* comes from the declarations of the specification under analysis.

From a non-terminal node, the left branch indicates that the formula is satisfiable and the right branch indicates that the formula is unsatisfiable. Terminal nodes represent one of two possible outcomes:

- **warning** signals that the formula of the root node may be vacuously satisfiable or vacuously unsatisfiable and the feedback to the original query is possibly misleading;
- $apply(A)_T$ indicates that the pattern for formula A should be applied where further diagnosis is required (T carries type details for free variables in A).

A question annotating a terminal node (e.g. *Q: Why is A valid?*) gives the user a context for the further application of patterns. Feedback from each query along a path through the patterns is noted, so that as much diagnostic information as possible is gathered for the original query. AA provides instantiations of a specification each time a query is satisfiable, i.e. each left branch of a tree produces an assignment for the variables of the specification and the queried formula.

Atomic expressions. An atomic expression is one of the two base cases (the other being the issue of a warning) in the application of a pattern. An atomic expression in Loy is a well-formed boolean expression that contains no logical connective. If we check the satisfiability of an atomic expression with AA we will be provided with an instantiation of the specification that satisfies the atomic expression, if the expression is satisfiable, and nothing if the expression is unsatisfiable. As long as the core constraints of the specification are consistent there is little scope for vacuity in the feedback to such a query. But whether an atomic expression is unsatisfiable, valid or neither can be discovered by testing the negation of the expression. For example, if p is satisfiable and $\neg p$ is unsatisfiable, then p is valid.

Pattern 1: Negation. (Figure 2.) The pattern for negation allows the validity of a formula to be queried by checking whether the negation of the formula is satisfiable. For example, to check whether A is valid, the satisfiability of $\neg A$ is checked. If $\neg A$ is satisfiable, A is not valid. But $\neg A$ may be vacuously satisfiable, so the satisfiability of A should be checked by applying the pattern for A. On the other hand, if $\neg A$ is unsatisfiable, A is valid. But the validity of A can be further investigated by applying its pattern.

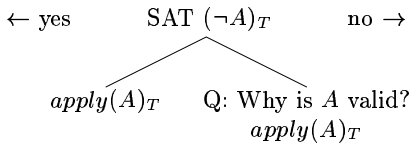

Fig. 2. Pattern 1: Negation

Pattern 2: Conjunction. (Figure 3.) The pattern for conjunction decomposes a formula into its conjuncts to identify either whether the formula is vac-

uously satisfiable or, if it is unsatisfiable, which conjuncts contain inconsistencies. For example, when checking the consistency of a method specification, the conjunction of the precondition, postcondition and frame condition is checked against the core constraints. Here, applying this pattern could uncover a vacuously satisfiable (and hence useless) postcondition or identify which part of the method specification contains an inconsistency. The pattern is applied by checking the satisfiability of a conjunction $A_1 \wedge \ldots \wedge A_n$. If $A_1 \wedge \ldots \wedge A_n$ is satisfiable, each conjunct is satisfiable. But $A_1 \wedge \ldots \wedge A_n$ may be vacuously satisfiable, so the pattern for each conjunct A_i is applied since, if A_i is vacuously satisfiable, so is $A_1 \wedge \ldots \wedge A_n$. On the other hand, if $A_1 \wedge \ldots \wedge A_n$ is unsatisfiable, at least one conjunct is unsatisfiable. Which conjunct or conjuncts are unsatisfiable could be established by applying the patterns for all combinations of conjunct $A_i \wedge \ldots \wedge A_j,\ 1 \leq i \leq j \leq n$.

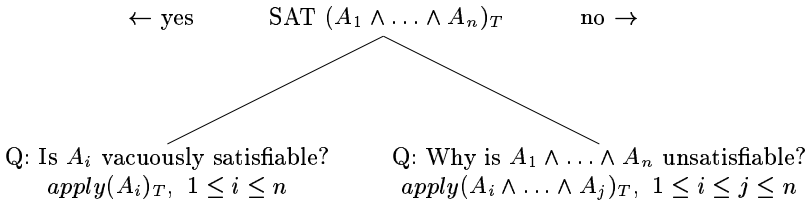

Fig. 3. Pattern 2: Conjunction

Pattern 3: Disjunction. (Figure 4). The pattern for disjunction decomposes a formula into its disjuncts to identify either whether the formula is vacuously satisfiable or, if it is unsatisfiable, why all disjuncts are unsatisfiable. The pattern is applied by checking the satisfiability of a disjunction $A_1 \vee \ldots \vee A_n$. If $A_1 \vee \ldots \vee A_n$ is satisfiable, at least one disjunct is satisfiable. But, as for Pattern 2, it is established whether $A_1 \vee \ldots \vee A_n$ is vacuously satisfiable by applying the pattern for each disjunct A_i since, if A_i is vacuously satisfiable, so is $A_1 \vee \ldots \vee A_n$. On the other hand, if $A_1 \vee \ldots \vee A_n$ is unsatisfiable, no disjunct is satisfiable. The pattern for each A_i is applied to investigate its unsatisfiability.

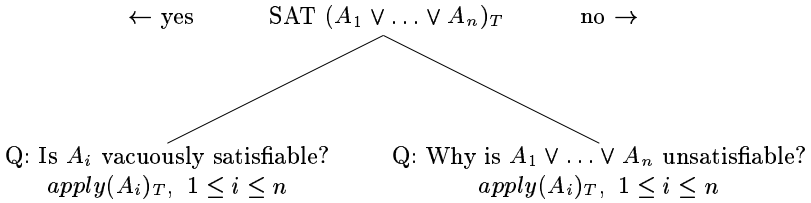

Fig. 4. Pattern 3: Disjunction

Pattern 4: Implication. (Figure 5.) The pattern for implication exposes whether or not an implication is vacuously satisfiable because of an unsatisfiable antecedent or valid consequent. For example, if A is the precondition for a

method and B the postcondition, we might ask whether $A \Rightarrow B$ for all models of the specification. If the precondition is unsatisfiable the tool will suggest that $A \Rightarrow B$ may be valid because it cannot satisfy $\neg(A \Rightarrow B)$. The pattern is applied by checking the satisfiability of an implication $A \Rightarrow B$. If $A \Rightarrow B$ is satisfiable, the satisfiability of both A and $\neg B$ is checked. If either A or $\neg B$ is unsatisfiable, $A \Rightarrow B$ is vacuously satisfied and a warning is issued. Otherwise, the subformulae A and B may be further investigated by applying their patterns. On the other hand, if $A \Rightarrow B$ is unsatisfiable, A must be valid and B must be unsatisfiable. Again, the subformulae may be investigated further by applying their patterns.

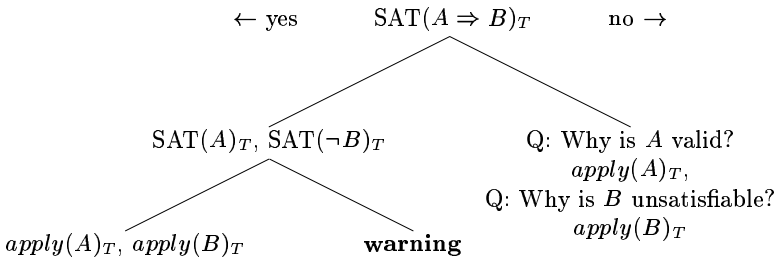

Fig. 5. Pattern 4: Implication

Pattern 5: Universal Quantification. (Figure 6.) Since the logic of Loy (and Alloy) is sorted, a universally quantified formula may be vacuously satisfiable because the domain ranged over by the quantifier is empty. For example, if a class specification C is inconsistent, then the formula $\forall\ c \in C \cdot A$ will be vacuously satisfied because C is unsatisfiable and hence there are no instances c. The pattern is applied by checking the satisfiability of a universally quantified formula $\forall\ x \in X \cdot A$. If $\forall\ x \in X \cdot A$ is satisfiable, the domain X is checked to see if it is empty. If it is, a warning is issued because $\forall\ x \in X \cdot A$ is vacuously satisfiable. If the domain is not empty, the satisfiability of A for all $x \in X$ is checked by applying the pattern for $A(x)$ for an arbitrary x (adding the binding $\langle x, X \rangle$ to T). On the other hand, if $\forall\ x \in X \cdot A$ is unsatisfiable, it should be established for what assignments to x A is unsatisfied. $\neg A$ is satisfied for at least one value of x but its satisfiability is checked (adding the binding $\langle x, X \rangle$ to T) simply to acquire a model of the specification that contains a counterexample to $\forall\ x \in X \cdot A$. The satisfiability of A for any assignment to x can be investigated further by applying the pattern for A (adding the binding $\langle x, X \rangle$ to T).

Pattern 6: Existential Quantification. (See Figure 7.) An existentially quantified formula may be vacuously unsatisfiable because the domain ranged over by the quantifier is empty. For example, similarly to the pattern for universal quantification, if a class specification C is inconsistent, then the formula $\exists\ c \in C \cdot A$ will be vacuously unsatisfied because C is unsatisfiable and hence there are no instances c. The pattern is applied by checking the satisfiability of an existentially quantified formula $\exists\ x \in X \cdot A$. If $\exists\ x \in X \cdot A$ is satisfiable, it is investigated whether A is vacuously satisfiable for an assignment to x by applying the pattern for $A(x)$ for an arbitrary x (adding the binding $\langle x, X \rangle$ to T). On the

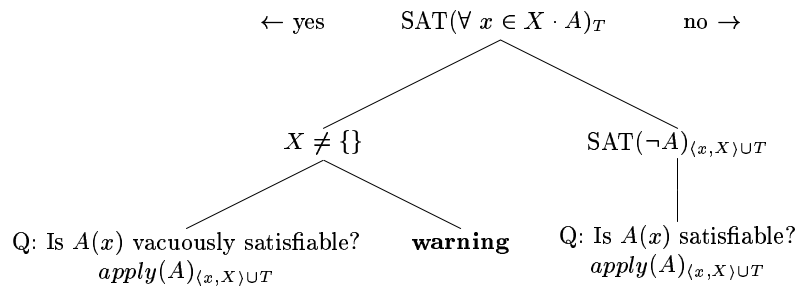

Fig. 6. Pattern 5: Universal Quantification

other hand, if $\exists\ x \in X \cdot A$ is unsatisfiable, the domain X is checked to see if it is empty. If it is, a warning is issued because $\exists\ x \in X \cdot A$ is vacuously unsatisfiable. If the domain is not empty, the satisfiability of A for all assignments to x is checked by applying the pattern for A (adding the binding $\langle x, X \rangle$ to T).

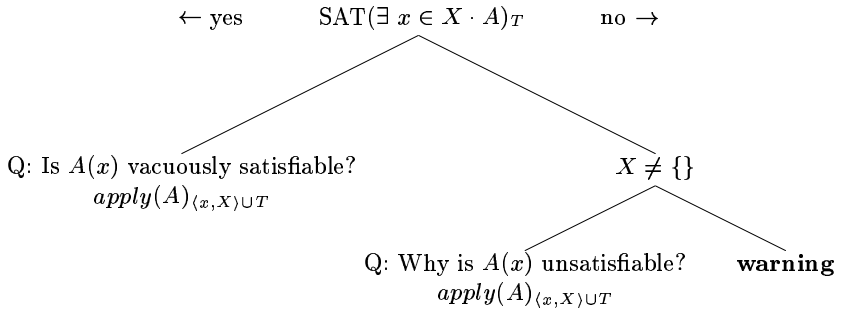

Fig. 7. Pattern 6: Existential Quantification

Examples. This Section ends with two examples. Let S be a specification, p, q and r be atomic expressions and C be a consistent class specification in S (i.e. C is a nonempty domain when used as a logical sort). Let the formula $\forall\ c : C \cdot P(c) \implies (Q(c) \lor R(c))$ be an invariant of C. With respect to a specification S, $P(c)$ is unsatisfiable, $Q(c)$ is valid and $R(c)$ is satisfied in some models of S and unsatisfied in others. The formula is satisfiable because C is consistent but it should be established whether or not it is vacuously satisfiable. Application of the pattern for this formula follows the path shown below.

Pattern 5 – Universal quantification.
$\forall\ c : C \cdot P(c) \Rightarrow (Q(c) \lor R(c))$ .. satisfiable
|
$C \neq \{\}$ .. satisfiable
|
Pattern 4 – Implication.
$P(c) \Rightarrow (Q(c) \lor R(c))$ .. satisfiable, $\langle c, C \rangle$
|

$P(c)$ .. unsatisfiable, $\langle c, C \rangle$

|

warning (unsatisfiable antecedent)

The invariant of C is entailed by the specification simply because $P(c)$ is unsatisfiable. But if, in fact, $P(c)$ was wrongly assumed to be satisfied in some models of S, the original feedback from AA, which simply suggested the property may be valid, would be inadequate.

To give an applied example, the patterns can be used to question the feedback given when checking the satisfiability of the specification of the method *assign* in the class specification for *Employee* (see Example 2, Section 3). The satisfiability of the method specification is checked by checking the satisfiability of the precondition, postcondition and frame condition with respect to the class specifications for *Employee* and *Project*, which is the type of the parameter p. For brevity, let $P \wedge Q \wedge R$ denote the conjunction of the precondition, postcondition and frame condition. The application of the pattern for this formula (the pattern for existential quantification) follows the short path shown below, which terminates with a warning that the formula is vacuously unsatisfiable.

Pattern 6 – Existential quantification.

$\exists\, e : Employee \;\exists\, p : Project \cdot P \wedge Q \wedge R$ .. unsatisfiable

|

$Employee \neq \{\}$ .. unsatisfiable

|

warning (specification for *Employee* is unsatisfiable)

The domain *Employee* is empty because the specification of *Employee* contains an inconsistency and cannot be instantiated. However, the clash between the two invariants is noted and the invariant for *Project* is removed. The query is run a second time and AA suggests that the property is consistent and provides an instantiation of the specification and an assignment to the variables e and p.

5 Encoding Loy in Alloy

This Section illustrates the encoding of Loy in Alloy. The encoding both gives Loy a semantics and allows automation of the patterns to be implemented on top of AA. We define the encoding of Loy in Alloy through the *encoding function*

$$\tau_{\nu} \;:\; \mathcal{L}_{\text{Loy}} \rightarrow \mathcal{L}_{\text{Alloy}}$$

that maps a Loy specification to an Alloy specification such that the models of a Loy specification are the models of that specification encoded in Alloy. That is, for a Loy specification s, $\mathsf{mod}\ (s) = \mathsf{mod}\ (\tau_{\nu}\ (s))$. The meaning of a Loy specification is thus the meaning of the Alloy specification that encodes it. The formal definition of the encoding function is omitted from this paper but the main features of the encoding are introduced below using the encodings of *Employee* and *ManagedEmployee*. All encodings are generated automatically.

A Loy class specification is encoded as an Alloy signature paragraph and set

of predicate paragraphs, one for each invariant, precondition, postcondition and frame condition of the class specification. The encoding of the field declaration and depends clause of *ManagedEmployee* (see Example 3) is shown below.

```
sig ManagedEmployee extends { manager : lone Manager } {
    // depends manager <- project
    idxOf (fields, manager) -> idxOf (fields, project) in depends
    # depends = 1
}
fact ManagedEmployee_fieldtable {
    let idx0 = Ord_/Ord.first, idx1 = Ord_/next (idx0) {
        all o : ManagedEmployee {
            at (o.fields, idx0) = o.manager
            at (o.fields, idx1) = o.project
}}}
```

The field declaration of the class specification is encoded as a field declaration of the signature paragraph. For example, the declaration *manager : Manager* in the Loy specification becomes the declaration *manager : lone Manager* in a signature paragraph *ManagedEmployee*. The modifier *lone* indicates that the field *projects* may be empty, capturing the default semantics of the Loy declaration. Every signature that encodes a class specification inherits the fields *fields* and *depends* from a root signature *Obj*, shown below.

```
sig Obj { fields : Seq [Obj], depends : SeqIdx -> SeqIdx }
```

The field *fields* is a sequence representing the fields of the class specification encoded by a signature that allows us to refer to the index of a field, i.e. the *location* of a field in a class rather than its value for a particular instance. The fact paragraph *ManagedEmployee_fieldtable* represents the field table (an Alloy sequence) for *ManagedEmployee*. The field *depends* is a binary relation that represents the depends relation of the class specification encoded by a signature. A depends clause is encoded as a constraint that specifies that *depends* contains pairings of the field on the left of the arrow with each of the fields on the right. For example, *depends manager <- project* is encoded as the constraint that the index of *manager* and the index of *project* are a pair in the *depends* field of *ManagedEmployee*. Further, since it is known that no class specification extends *ManagedEmployee* for this encoding and, therefore, that the depends relation will not be added to, its cardinality is declared to be 1 (*# depends = 1*) to prevent erroneous extra pairs being inserted by AA during analysis.

An invariant of a class specification is encoded as a predicate paragraph containing a formula that applies the invariant to all instances of the signature that encodes the class specification. For example, *ManagedEmployee* inherits from *Employee* the invariant *no project.manager* (see Example 2), which is encoded as the predicate paragraph shown below.

```
pred Employee_I () { all x : Employee | no x.project.manager }
```

Encoding an invariant as a predicate paragraph allows analyses of a specification to be carried out with and without the invariant present. Specifically, this allows us to check the consistency of the invariant itself with respect to the rest of a specification.

To represent the concept of state implicit in a Loy specification the signature paragraphs *Id* and *State* are created. The *Id* and *State* signatures for the encoding of our example specification are shown below.

```
sig Id { }
sig State {
    manager : Id lone -> lone Manager,
    project : Id lone -> lone Project,
    employee : Id lone -> lone Employee,
    managedEmployee : Id lone -> lone ManagedEmployee
}
```

The fields of the signature *State* are partial mappings from a set of references (*Id*) to sets of instances of the signatures that encode the class specifications. These mappings represent persistant references to instances of a class specification across states. For example, given two states, s and s', and an *Id*, i, we can refer to an instance of *ManagedEmployee* in s and s', respectively, with the expressions *s.managedEmployee[i]* and *s'.managedEmployee[i]* where i provides a reference to an instance across states. Strictly, *s.managedEmployee[i]* and *s'.managedEmployee[i]* denote different instances of the Alloy signature *ManagedEmployee* that represent (possibly) different values of a single instance of the Loy class specification *ManagedEmployee*.

A method specification is encoded as three predicate paragraphs, one each for the precondition (given a P suffix), postcondition (given a Q suffix) and frame condition (given an F suffix). The encoding of the method specification for *assign* in *ManagedEmployee* is shown below.

```
pred ManagedEmployee_assign_P (i : Id, s0 : State, p : Project) {
    no s0.managedEmployee[i].project
}
pred ManagedEmployee_assign_Q (i : Id, s0, s1 : State, p : Project) {
    s1.managedEmployee[i].project = p
    s1.managedEmployee[i].manager = p.manager
}
pred ManagedEmployee_assign_F (i : Id, s0, s1 : State, p : Project) {
    let o = s0.managedEmployee[i], o' = s1.managedEmployee[i] {
        all k : Seq/SeqIdx {
            at (o.fields, k) = at (o'.fields, k) ||
            k = idxOf (o.fields, o.project)
        }
        all x : ID {
            s1.managedEmployee[x] = s0.managedEmployee[x] || x = i
```

```
                s1.manager[x] = s0.manager[x]
                s1.project[x] = s0.project[x]
                s1.employee[x] = s0.employee[x]
}}}
```

Special parameters representing a reference (i) and before and after states (*s0* and *s1*) are added to the parameter p of *assign*. Unprimed and primed variables of the Loy specification, respectively denoting before and after state values, are then encoded as expressions that refer to instances of a signature relative to *s0* or *s1* through the persistent reference *i*. For example, *project* and *project'* are encoded as *s0.managedEmployee[i].project* and *s1.managedEmployee[i].project*.

A frame condition is constructed from a modifies clause. The mappings represented by the fields of *s0* and *s1* are specified to be the same except possibly where a field appears in the modifies clause, permitting the method to alter its value. The after state has to be constructed from the before state explicitly. Consider the variable expression $v_1 \cdot \ldots \cdot v_n$. If this expression appears in the modifies clause of a method specification, an invocation of the method is permitted to change the value of v_n. But, because of the value semantics of Alloy, if the value of v_i changes (for $1 < i \leq n$), then the value of v_{i-1} must also be updated. Further, for each v_i, the fields that are not permitted to change must be constrained to have the same value in *s0* and *s1*. Finally, the values of the fields of *s1* are constrained: only the mappings for types of fields that are permitted to change may differ from those of *s0*. For example, since *assign* only changes the value of its receiver, which is an instance of *ManagedEmployee*, only the mapping from *Id* to *ManagedEmployee* may differ between *s0* and *s1*.

The queries represented by the tree nodes in the patterns of analysis are implemented as Alloy queries to AA. The formulae are encoded in Alloy and wrapped in predicate paragraphs. The type information carried around in the set T becomes the parameter declarations for this predicate paragraph. For example, the query $\mathrm{SAT}(A \Rightarrow B)_{\langle x,X\rangle,\langle y,Y\rangle}$ is encoded as the Alloy predicate paragraph

```
pred (x : X, y : Y) { A implies B }
```

and submitted to the tool as a consistency query. The feedback from AA for each query is then interpreted by the user in terms of the Loy specification under analysis.

6 Related Work

Specification animation such as that supported by the ProB [16], Bogor [22] and JML-TT-Animator [14] tools is a simple way to check basic properties of specifications, as inconsistency may be caught quickly in the exploratory environment provided by an animator. However, as with test-driven development the onus is on the practitioner to direct the testing in directions that catch the errors. AA enhanced by patterns of analysis removes much of this onus and guides the

practitioner through an exhaustive automated analysis of a specification.

The Analysable Annotation Language (AAL) [13] is a notation that borrows from both Alloy and JML and is designed to annotate Java programs with JML-like specifications. An encoding of AAL in Alloy allows these specifications to be checked by AA. However, AAL focuses more on the checking of an implementation against a specification and not on the preliminary analysis of the specification itself. Alternate encodings of state in Alloy are presented in [12], which encodes state in order to use Alloy to support software testing, and [18], which introduces VAlloy, an extension of Alloy that adds virtual functions and a simple state model to the language. However, neither encoding gives a full treatment of specifications that handle frame conditions or depends clauses.

Finally, a translation from B AMN to Alloy is presented in [20]. The B toolset provides limited support for automatically discharging proof obligations and many proofs have to be constructed interactively. By complementing a theorem prover with AA, using the latter to check the consistency or validity of a property before attempting a proof, valuable confidence that the property is in fact provable is gained. Though this is another example of AA being invoked to support specification analysis, our approach addresses the need for a lightweight technique that provides immediate returns to the software developer.

7 Conclusion

In order to make a lightweight formal technique even more attractive to commercial software developers, we have defined a lightweight formal object-oriented specification language that is supported, through an encoding into Alloy, by AA. We have also identified patterns of analysis that guide a developer through the analysis of a specification to establish its satisfiability before implementation.

We have presented the patterns in the context of AA but the patterns are intended to be generally applicable in any analysis environment that depends on querying properties for satisfiability with respect to a model. Automatic generation of an Alloy encoding from a Loy specification has been implemented but a full implementation of the approach is still to be completed. The patterns will be implemented on top of AA such that a satisfiability query submitted to the tool for a given formula will immediately invoke the pattern for that formula so that misleading feedback is never given.

A current limitation of this approach is the time it takes to run several satisfiability queries in a row. Most desktop machines take a couple of minutes to compute a single pattern before feedback is given, even for an Alloy encoding of a couple of hundred lines. One way to shorten computation time would be to couple AA with a version control system in order to keep track of which parts of a specification were known to be consistent with each other and then only recheck the satisfiability of properties that have been affected by editing since the last analysis. Once an initial check had been performed, this would greatly reduce the computation that needed to be done in the application of each pattern. Further research will also include a detailed study of the complexity and completeness of the analysis patterns proposed.

References

1. K. R. Apt. Ten Years of Hoare's Logic: A Survey – Part 1. *ACM TOPLAS*, 3(4):431–483, 1981.
2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec♯ Programming System: An Overview. In *CASSIS Post-Proceedings*, 2004.
3. L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An Overview of JML Tools and Applications. *Software Tools for Technology Transfer*, 2005.
4. S. Caulkin. Software Must Stop Bugging Us. *The Observer, Sunday March 7*, 2005.
5. Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards. Model Variables: Cleanly Supporting Abstraction in Design By Contract. Technical Report 03-10b, Department of Computer Science, Iowa State University, August 2004.
6. E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, 1996.
7. R. W. Floyd. Assigning Meanings to Programs. In J. T. Schwartz, ed, *Mathematical Aspects of Computer Science*. American Mathematical Society, 1967.
8. C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.
9. D. Jackson. Alloy 3.0. `http://alloy.mit.edu/beta/reference-manual.pdf`.
10. D. Jackson. Automating First-Order Relational Logic. In *FSE'00*, 2000.
11. D. Jackson and J. Wing. Lightweight Formal Methods. IEEE Computer, April 1996.
12. S. Khurshid. Generating Structurally Complex Tests from Declarative Constraints. PhD Thesis, MIT, 2004.
13. S. Khurshid, Darko Marinov, and Daniel Jackson. An Analyzable Annotation Language. In *OOPSLA'02*, pages 231–245. ACM Press, 2002.
14. Laboratoire d'Informatique de l'université de Franche-Comté. JML-Testing-Tools: JML Animator. `http://lifc.univ-fcomte.fr/~jmltt/index.php?lang=en`.
15. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Technical Report 98-06z, Department of Computer Science, Iowa State University, June 2004.
16. M. Leuschel and M. Butler. ProB: A Model Checker for B. In *FME 2003: Formal Methods*, pages 855–874. Springer-Verlag, 2003.
17. B. H. Liskov and J. M. Wing. A Behavioral Notion of Subtyping. *ACM TOPLAS*, 16(6):1811–1841, November 1994.
18. D. Marinov and S. Khurshid. VAlloy - Virtual Functions Meet a Relational Language. In *FME 2002: Formal Methods - Getting IT Right*, pages 234–251. Springer-Verlag, 2002.
19. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.
20. L. Mikhailov and M. Butler. Combining B and Alloy. Technical Report DSSE-TR-2001-2, Declarative Systems and Software Engineering Group, Department of Electronics and Computer Science University of Southampton, February 2001.
21. P. Müller and A. Poetzsch-Heffter and G. T. Leavens. Modular Specification of Frame Properties in JML. *Concurrency, Computation Practice and Experience*, 15:117–154, 2003.
22. Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An Extensible and Highly-Modular Software Model Checking Framework. In *ESEC/FSE'03*, pages 267–276. ACM Press, 2003.