Precise Specification of Design Pattern Structure and Behaviour

Ashley Sterritt, Siobhán Clarke and Vinny Cahill

Lero@TCD,
Distributed Systems Group,
Trinity College Dublin
{firstname.lastname}@scss.tcd.ie

Abstract. Applying design patterns while developing a software system can improve its non-functional properties, such as extensibility and loose coupling. Precise specification of structure and behaviour communicates the invariants imposed by a pattern on a conforming implementation and enables formal software verification. Many existing design-pattern specification languages (DPSLs) focus on class structure alone, while those that do address behaviour suffer from a lack of expressiveness and/or imprecise semantics. In particular, in a review of existing work, three invariant categories were found to be inexpressible in state-of-the-art DPSLs: dependency, object state and data-structure. This paper presents Alas: a precise specification language that supports design-pattern descriptions including these invariant categories. The language is based on UML Class and Sequence diagrams with modified syntax and semantics. In this paper, the meaning of the presented invariants is formalized and relevant ambiguities in the UML Standard are clarified. We have evaluated Alas by specifying the widely-used Gang of Four pattern catalog and identified patterns that benefitted from the added expressiveness and semantics of Alas.

1 Introduction

Object-oriented design patterns 'capture design experience in a form that people can use effectively' [1] to develop software with improved non-functional properties such as re-usability, extensibility and loose coupling. Design patterns (later referred to as patterns) dictate certain relationships between classes and objects such as inheritance, object composition, delegation and information hiding. In a pattern implementation, the actor that performs an action is important, in contrast to an algorithm, which may be implemented by any combination of actors. Thus, patterns define object-oriented protocols that must be followed in an implementation. Pattern specifications define a number of roles, most of which are mutually exclusive, to be filled by actors (classes, objects or methods) in the implementation. Precise specification of pattern structure and behaviour communicates the invariants imposed by a pattern on a conforming implementation and enables accurate formal software verification, by, for example, avoiding false

positives due to specifications that are too generic.

We performed an analysis of the widely-used Gang of Four (GoF)[1] pattern catalog from which we identified five invariant categories that were used to classify existing work in the area. These categories are cardinality, dependency, control flow, object state and data structure.

Design patterns place constraints on multiple entities (objects, classes and inheritance hierarchies) and are also more generic than concrete software architectures as they describe interactions between entities, whose number and type are unknown. Patterns thus present a subtly different specification challenge. Patterns such as the Abstract Factory and Visitor patterns place constraints on the relation between the number of entities (classes and methods) occuring in separate inheritance hierarchies. For example, in the Visitor pattern, each ConcreteVisitor should have a `visit` method for each ConcreteElement in the Element hierarchy. We refer such invariants as *cardinality* invariants.

The key invariant of numerous GoF patterns, for example the Façade and Abstract Factory patterns, can be expressed informally as "Class A should not be directly associated with Class B" or "Class A shouldn't be hard-coded to use a particular subclass of Class B". The first of these informal statements refer to the static type of variables and has been termed *interface dependency*, while the second refers to the creation of instances of one class by another and is termed *implementation dependency*.

Object state invariants concern the runtime values of objects and their attributes such as whether they have been initialized or not, the equality of attributes and object identity. The Memento pattern's intent is 'to capture and externalize an object's state so that the object can be restored to this state later. [1]' Thus, an invariant on a Memento implementation is that a subset of the state of the Memento is in some relation (e.g., equality) to the state of another object (the Originator), at a particular point in the execution (Memento creation). Also, the state of the objects should remain in this relation until some other execution point (some undo operation). This particular invariant sub-category is called *inter-object state dependency. Control flow* invariants are defined as invariants that place constraints on the control-flow in a pattern. Relevant control flows are sequencing, method calls, conditionals and loops.

A number of GoF design patterns describe the use of or are often applied to user-defined recursive *data structures* that are required to demonstrate properties such as being cycle free or not containing elements that are shared (i.e., have two distinct objects that hold references to it). The Composite pattern, for example, "composes objects into tree structures to represent part-whole hierarchies"[1]. In most realizations of the Composite pattern, sharing of sub-trees or leaves is prohibited as this complicates traversal or violates the tree's semantics.

In the Decorator pattern, adding a new Decorator to the Decorator chain should not make the Decoratee object unreachable.

Three of these five categories were found to be insufficiently addressed (i.e., inexpressible or ambiguously defined) by the state-of-the-art DPSLs. The three insufficiently addressed categories are (implementation) dependency, object state and data structure. This paper presents Alas (Another Language for pAttern Specification): a precise specification language that supports design-pattern descriptions including the three invariant categories discussed above. The language is based on UML Class and Sequence diagrams with modified and extended syntax and semantics. UML is the de facto standard for object-oriented software modelling, and UML Sequence diagrams provide a suitable level of granularity at which to describe the inter-object protocols imposed by design patterns. Nevertheless, a number of syntax extensions and clarifications of existing concepts are required for UML to be suitable for precise specification of design patterns enabling formal software verification. In the current version of Alas, there is no concurrency: it can describe only sequential programs. This choice was made to simplify the initial design and the planned supporting verification tool. We plan to add support for concurrency in a future version.
í While we chose UML as a basis for Alas, UML in its current form was considered unsuitable for precise design-pattern specification for a number of reasons. Le Guennec et al. [2] notes that UML, despite its templates and parameterized binding, is not suited to expressing cardinality invariants. This is due to a lack of control over the number of bindings that can be made between classes and roles. In addition, patterns require logical statements to be made about software structure, making it necessary to use the Object Constraint Language (OCL) [3] at the meta-model level to define non-standard, pattern-specific entities. Many constructs (such as the CombinedFragments newly introduced in UML 2.0) are described too informally to be the basis for software verification, where precise semantics are required. Also, as design patterns are generic solutions that can be applied in many different contexts, their specifications need to mirror this genericity in some ways that are not supported in UML. One example occurs when one object's value should be some function of another objects, but this function is not common to all pattern variants. The 'reflects' keyword, introduced in Contracts [4] can be used to express this abstract state dependency.

When placing invariants on the state of interacting objects, it is necessary to distinguish between aliases (two names that refer to the same object) and copies (two objects with identical values). As discussed in Section 3, UML and OCL are vague with regard to this distinction. Alas defines binary object predicates `isAlias` and `isCopy` to resolve this ambiguity. As OCL lacks an operation to express transitive closure, expressing these data-structure 'shape' invariants in OCL would be verbose and error-prone. For example, the user must be careful to write constraints that do not go into infinite cycles and are undefined. Alas contains basic transitive operations on recursive data structures and uses these

to define shape properties such as heap-sharing (two distinct objects holding references pointing to the same object), the existence of cycles and reachability.

In UML Sequence diagrams, a Lifeline represents one and only one object: "While Parts and StructuralFeatures may have multiplicity greater than 1, Lifelines represent only one interacting entity... If the referenced ConnectableElement is multivalued... then the Lifeline may have an expression (the selector') that specifies which particular part is represented by this Lifeline. [5]" This prevents the user from specifying the common case of a method being invoked on each element of an unbound collection in turn. We formalize an existing idiom for expressing this case, by relaxing the binding semantics under particular conditions, allowing them to represent different objects at different times.

The structure of the remainder of this paper is as follows: Section 2 discusses related work. Section 3 introduces Alas through examples relating to the invariant categories described in this paper. Section 4 defines a precise meaning for each of the non-standard extensions and clarifications provided in Alas. Section 5 evaluates the expressiveness of Alas compared to the state-of-the-art DPSLs with respect to the GoF catalog. Finally, Section 6 concludes and considers some directions for future work.

2 Related Work

Numerous DPSLs choose to focus on a pattern's 'essence' or 'leitmotif', specifying design pattern structure that is thought to be common to all pattern variants. These approaches are typically also capable of expressing cardinality invariants. LePUS [6] defines a graphical notation for expressing sets of classes in an inheritance hierarchy and sets of associated methods, along with relationships between them such as invocation and creation. This allows cardinality constraints to be specified simply, graphically and precisely, as it is based on higher-order logic. Le Guennec et al. [2] and Mak et al. [7] handle cardinality invariants using a UML Profile that introduces multiplicities in UML Collaborations at the metamodel level. Lauder and Kent [8] introduce a fourth compartment into the UML Class syntax that utilizes their constraint diagrams, which are also based on set semantics. However, each of these DPSLs, by focusing on structure only, completely ignore the behaviour required to satisfy a pattern's intent.

In RSL [9], a renaming map is used to associate entities in patterns to their corresponding implementation entities, supporting cardinality as well as implementation dependency invariants. The specifications in RSL, however, are verbose and implementation-oriented, making the intent of the pattern hard to understand without significant effort. Lano et al. [10] formally specify patterns in detail, including behaviour such as method calls and object creation, and define a refinement relationship between a software program before and after applying

a pattern. The refinement proof must be performed manually though, and this is challenging, given the mathematical basis of the language. BPSL [11] supports the specification of the structure and behaviour of patterns, using first-order logic and the Temporal Logic of Actions (TLA), respectively. The structural part has a similar expressiveness to UML Class diagrams while the behavioural part describes some object state properties such as value equality of variables at a particular execution state. As only one pattern specification is presented, it is difficult to assess the applicability of the language to patterns in general. Dong et al. [12] also utilize TLA for pattern behaviour specification with some precise implementation conformance rules, but with less expressiveness overall.

RBML [13] and FUJABA [14] use UML 2.0 Class and Sequence diagrams to specify pattern structure and behaviour, such as method calls, conditionals and loops. RBML also provides role realization multiplicities to describe cardinality invariants, and is thus one of the most expressive DPSLs overall. It does not address object state or data structure invariants, however, and makes no effort to define a precise semantics for the language. In summary, of the five invariant categories we identified, cardinality and control-flow are well supported in the literature, dependency has had some attention while there are significant gaps in the support for object-state and data-structure invariants.

3 Pattern Specification in Alas

In Alas, pattern specifications are made up of structural diagrams and behavioural diagrams. Alas structural diagrams are UML Class diagrams augmented with first-order logic, ranging over structural entities (classes and methods), to support the specification of cardinality invariants. Interface dependency invariants are supported using binary class operators such as `hasRef` and `calls`, along with logical conjunction, disjunction and negation. Implementation dependency invariants are discussed in the following section.

Behavioural diagrams in Alas are based on UML 2.0 Sequence diagrams, currently making use of only the `alt`, `opt` and `loop` CombinedFragments to express control-flow invariants. Object-state invariants are placed in constraint boxes that are connected to particular points in the control flow. These boxes can contain standard OCL collection operators such as set intersection and union. Non-standard extensions allow the expression of, for example, inter-object state dependency and data structure invariants.

3.1 Implementation Dependency Invariants

The dependency invariants in some patterns, including AbstractFactory, Prototype, Bridge and State, are more subtle than simply forbidding variables of a particular type in class definitions. A summary of their common intent might

be that "a client holds a reference to an object, but is not hard-coded to a particular implementation (subclass)." The Abstract Factory pattern "provide[s] an interface for creating families of... objects without specifying their concrete class." Thus, a client should never contain the code: `Maze aMaze = new Maze()` or `BombedMaze bMaze = ...` as the first performs the initialization itself, and the second commits to a particular subclass. Instead, creation of the object is delegated to a factory object. This is described in the Alas invariant below:

```
Client hasRef Product AND NOT (Client hasRef ConcreteProduct) AND
   NOT (Client isInitializer ConcreteProduct) .
```

where `ConcreteProduct` inherits from `Product`. `isInitializer` is a binary operator that states that the subject (first) operand, which may be a class or method, calls the second operand's constructor directly. The first two clauses state that the Client has a reference to the superclass (`Product`), but not to a particular set of subclasses of `Product`: those that inherit from `ConcreteProduct`. The third predicate states that the Client does not initialize any object that is a subclass of Product. Note that dependency predicates apply to a class and all its subclasses, unless over-ridden by other predicates, as shown above.

3.2 Object-State Invariants

The role of a Factory Method is to return a newly created instance of a Product class. Thus, a key invariant of the Factory Method pattern is that a new object is returned, i.e., the object created by the Product constructor is the same object that is returned by the Factory Method. A related creational pattern is the Prototype pattern, which avoids creating a new instance by copying a prototypical one. One invariant of the Prototype pattern is that the object returned by the Prototype's `clone()` method is *not* the same object as the prototype, but should have identical values for some subset of its state. Thus, to specify the Factory Method and Prototype patterns precisely, it is necessary to be able to express the concepts of object identity and value equality.

The OCL Standard ([3] Appendix A: Semantics, Section 2.2) suggests that the meaning of the equality operator, when applied to two object operands, is defined as value equality: "The equality of values of the same type can be checked with the operation $=_t$" (defined for all types) and indeed the implementation of Dresden-OCL's [15] equality operator calls the Java `equals()` method. Collection operators, which one might expect to be defined in terms of object identity, also seem to be value-based. Set subtraction, for example, is defined as "`S - <v>`: produces a Sequence equal to `S`, but with all elements equal to `v` removed." This potentially removes many objects with equal values, rather than a single object uniquely identified by `v`.

Object identity is discussed briefly in Appendix A, Section 1.2.1: "Objects are referred to by unique object identifiers" [3]. The set `oid(c)` is also defined as the set of object identifiers for a class. This set is not used in the definition of any of the relevant OCL operators, adding to the evidence that objects are compared by value. The UML Standard also makes little reference to object identity. A DataType is described as being "similar to a Class. It differs from a Class in that instances of DataType are identified only by their value." However, the meta-class Class has no attributes or associations that could be used to store identity and both Class and DataType occur at the same level of the UML meta-inheritance hierarchy, inheriting directly from Classifier, and nothing else.

Object identity and value equality are distinguished explicitly in Alas using the `isAlias` and `isCopy` binary operators respectively. These are defined precisely in terms of object identifiers and values in Section 4. The key invariant of the Factory Method pattern uses `isAlias`, and is shown in Figure 1. Note that in Alas conditions are connected at any branching or joining of control-flow. The connection position of the invariant in Figure 1 is equivalent to a post-condition in OCL. Conditions do not need to span multiple lifelines as there is no concurrency. The `clone()` method of the Prototype pattern also has an attached postcondition, specified as: `returnval isCopy prototype.this`. This condition identifies the `prototype` object and the newly-created and returned object as copies.

It is possible that two lifelines in the same diagram become bound to the same object in the implementation. If a lifeline is intended to identify one unique interacting entity, as suggested by the UML standard, then this binding is a violation of the pattern specification. Roles in the specification are thus always mutually exclusive. We have found that this creates a difficulty in specifying the Chain of Responsibility (CoR) pattern in the case that there is no default Handler for requests. The role of the object that creates the request and the object that handles the request could be the same object, though it is necessary to represent the two roles in two separate lifelines. For this reason, we have defined a n-ary operator `notMutEx` that specifies that two lifelines in a sequence diagram (or two classes in a structural diagram) are not required to be bound to different entities.

3.3 Control-Flow Invariants

In the Observer pattern, when an update occurs to the Subject's state, it calls its `notify` method. `Notify` iterates over the Subject's list of Observers, calling `Update` on each of them in turn. The specification of this behaviour is given in Figure 2, where the names of the loop variable and lifeline selector match. While this is an existing idiom used for describing interactions with entities that have an unbounded number of elements, it is non-standard for two reasons: it requires a redefinition of the immutable lifeline/object binding and the only valid loop

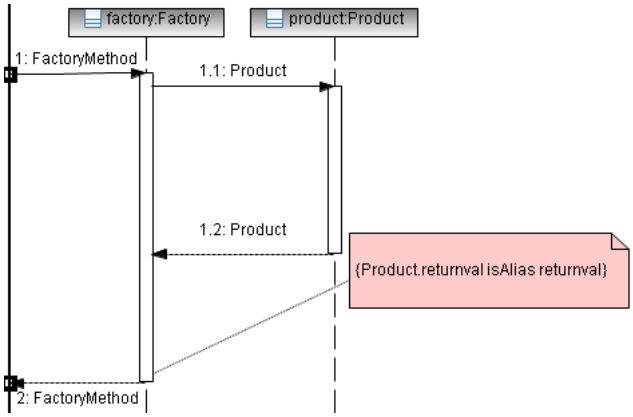

Fig. 1. Use of the `isAlias` predicate to specify object identity in the Factory Method specification

operands defined in the standard are `maxint` and `minint` or a boolean expression.

3.4 Data-Structure Invariants

To specify data-structure shape invariants, the specification language must be capable of expressing relations between the position of objects in a recursive structure. There is no primitive operator in OCL for expressing transitive closure directly and it is not discussed in the latest OCL Standard [3]. To obtain the transitive closure of a relation, the user may write a recursive function similar to:

```
allPredecessors = self.predecessor
  → union (self.predecessor.allPredecessors) .
```

This statement, however, may not have the desired effect, as it may go into an infinite loop if the data structure has cycles and would then evaluate to an undefined value. Some tools supporting OCL, such as Eclipse, provide a safe closure operation, by building a collection using an iterative fixpoint algorithm [16]. Also, in OCL queries and constraints, it is possible only to refer to objects that are navigable from the contextual object via associations. In a singly-linked list, for example, this corresponds to all the objects occuring later in the list than the contextual object. When defining data-structure properties, however, it is often more convenient to refer to an object's predecessors: whether heap-sharing occurs can be expressed succinctly by evaluating if the object has two or more immediate predecessors (see section 4). In OCL, it would be necessary to begin from the root of the structure and attempt to identify two (potentially very long) paths to the object.

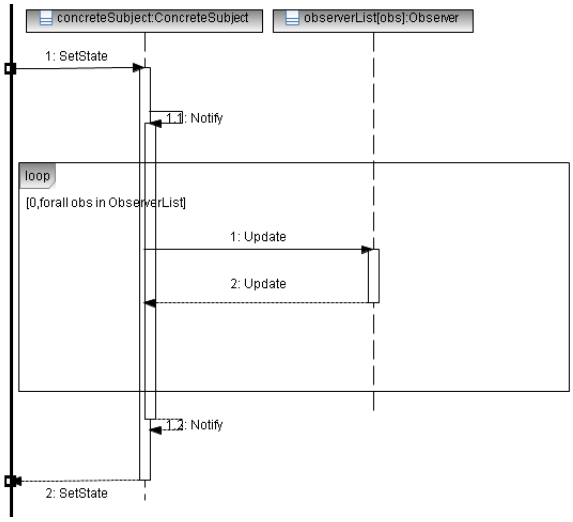

Fig. 2. Specification of the Subject's *Notify* method that involves iteratively calling each object in an unbounded structure

Alas data-structure predicates are defined in terms of transitive and non-transitive (one step) versions of the primitives `isPredecessor` and `isSuccessor`. This simplifies the definition of transitive closure relations when compared to OCL and allows for safer and more concise specification. The invariant of the Composite pattern is specified in Figure 3. Note that this invariant is sufficient to ensure the Composite tree is cycle-free, as long as `c` is the only Component added to the tree in the `add()` method and the `add()` method is the only method that adds to the tree. The first node in a cycle must have two predecessors, i.e. applying the `isShared` predicate to it would evaluate to true.

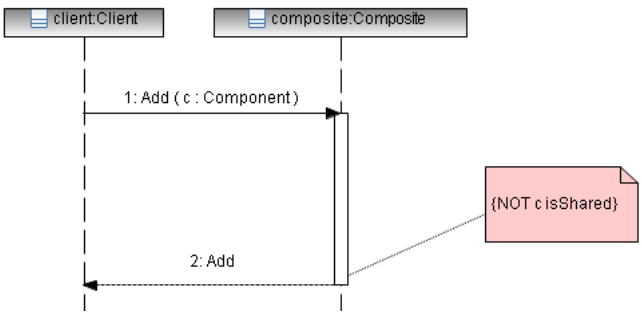

Fig. 3. Specification of a *Composite*'s *Add* method where sharing of nodes is forbidden

The CoR pattern decouples the sender and receiver of a request by creating a chain of objects, each of which has the option to handle the request or pass it on. A desirable property of the CoR pattern is that every request eventually gets handled by some Handler. This is often ensured by providing a root Handler that is the end of every chain of Handlers, which can provide some default response. This is specified in Alas using predicates as 'There exists a handler that is capable of providing a response to this request type and this handler is reachable from every other handler" (see Figure 4). The constraint box contains a first-order logic statement quantifying over each object in the Handler chain. The @ sign is an ASCII substitute for the first-order logic 'it holds that' from Z notation. The specification states that the role `default`'s `HandleRequest` method will always call its `ServeRequest` method, i.e., it will never forward the message without handling it. Note that Alas's default semantics is that a diagram specifies required and not optional behaviour (i.e., universally quantified paths). In this way, it is equivalent to a Sequence diagram placed entirely within an `assert` fragment. Diagrams with existential path quantifiers are outside the scope of this paper. Note, the definition of the `chain` data structure is omitted here, but currently data-structure definitions are done textually. These examples show that Alas data structure predicates allow sophisticated statements to be made about the recursive data structures in programs concisely that were previously inexpressible in the context of DPSLs.

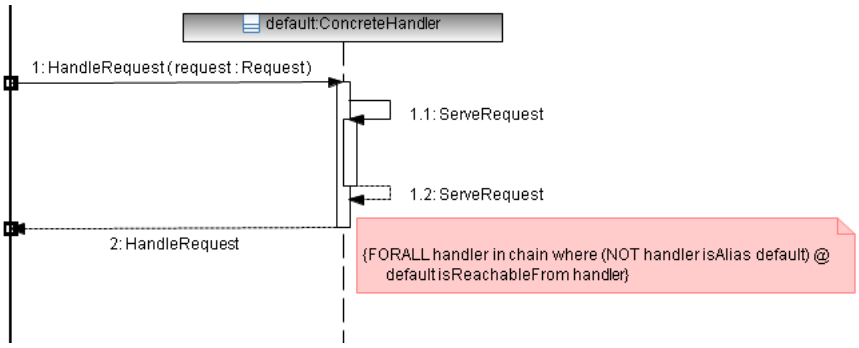

Fig. 4. Specification of the CoR pattern, where a *default* handler is the final node in a chain

4 Semantics

With limited space, this section provides definitions of only some of the more important syntactic elements introduced in the previous section. The meaning of dependency syntax elements is straight-forward and has been sufficiently

described in the previous section for an intuitive understanding. Control-flow invariants make use of the `alt` and `opt` CombinedFragments. These are only informally described in the UML Standard and could be interpreted as either mandatory or potential choice. In Alas they are interpreted as mandatory choice, following Lund and Stølen [17]. This means that an `opt` CombinedFragment, for example, in an Alas behavioural diagram indicates that a conditional statement should occur in the implementation. Potential choice (where a conditional in the specification indicates that the implementor can choose whether or not to implement the behaviour) is useful in pattern variant specification. A `variant` CombinedFragment, which has one or many compartments, each indicating an implementation option, is also being defined.

4.1 Object State

To define object state invariant syntax, the state of a program is represented as a transition system. In each state ($s \in S$), there is a set of objects (O) that can grow and shrink between states as objects are created and destroyed, but represent a fixed set in any one state. Each object has a unique identity, which can be accessed using the function $id(o)$. Each object has a set of attributes (A), each element of which is accessed using the notation $obj.a$, and also a subset of attributes CA (i.e., $CA \subset A$) that is considered when deciding if an object is a copy of another. CA is problem-specific and is defined by the user. The value of attributes in each state can be obtained using the function $Val(a)$. Each object is bound to a set of role names (N), and the function $obj(n)$ maps a role name to its object.

We can now define the Alas operators `isAlias` and `isCopy`:

$$\texttt{name isAlias otherName} \rightarrow_{def} obj(name) = obj(otherName) \ .$$
$$\texttt{name isCopy otherName} \rightarrow_{def} \forall ca : CA \bullet Val(obj(name).ca) = Val(obj(otherName).ca) \ .$$

For two objects to be copies of one another, they must be the same kind, but not the same type. Both operators are commutative and transitive. Each role in Alas behavioural diagrams is by default mutually exclusive, so given two role names $name$ and $otherName$ of the same kind, it holds that:

$$\neg \exists name, otherName \bullet obj(name) = obj(otherName) \ .$$

This can be over-ridden in Alas using `isAlias`, or the `notMutEx` n-ary operator, which has been used in the specification of the CoR pattern variant where there is no default handler.

4.2 Control Flow

The UML Standard implies that there is an immutable binding between a lifeline name in a specification and an object in a candidate implementation. A selector may identify an object in a fixed position in a structure and the absence of a selector leads to an arbitrary object being bound, but these bindings are still to single objects and immutable. It also limits the valid operands that may be used in loop fragments to `maxint` and `minint` or a boolean expression. This prevents the user from specifying interactions with structures of an a priori unknown (or mutable) size. More formally, for a transition system covering the part of the program referred to by the specification with initial state is and final accepting state fs and ordering relation $\geq$ ('happens before or simultaneous'), the standard interpretation is defined as:

$$\forall s1, s2 : S | (is \geq s1 \geq fs) \wedge (is \geq s2 \geq fs) \bullet obj(n, s1) = obj(n, s2) \ .$$

where $obj(name, state)$ is an extended version of the function defined in the previous section that maps a name in a particular state to an object. In Alas, when a selector is specified that is identical to the loop variable, this requirement is relaxed to allow rebindings to occur each time the object's lifeline returns the flow of control. For a specified call event transition ctA and its accompanying return event transition rtA, ctA and rtA can replace is and fs in the above constraint. After the object returns the flow of control, it releases its binding. After the loop variable is incremented, the lifeline is then free to be rebound to the new value of the variable (one greater than the previous value). The formalization of this idiom, and the corresponding extension of the allowed operands of the loop CombinedFragment enables the precise specification of interactions with unbounded data structures.

4.3 Data Structure

A recursive data structure is defined as a directed graph, where the nodes are objects (with unique identities) that may occur more than once in the structure and the edges are references labelled by their variable name. Null is a valid value for a node. The extent of the data structure stretches from some root node until all paths from the root encounter a null node. We define `hasSuccessor*` as a transitive binary operator taking two object operands that evaluates to true if it is possible to navigate along the direction of the references from the first operand to the second operand. `hasPredecessor*` is a similar operator, though it navigates in the opposite direction to the references (this operator distinguishes Alas from OCL in this context). Both operators have a non-starred counterpart, that indicates navigation is only performed for one step. Thus, `o hasSuccessor p` is true iff one of `o`'s immediate successors is `p`. Data-structure properties can be defined using these operators and first-order logic. Here, the definition of `isCycleFree`, `isReachableFrom` and `isShared` is shown:

$$
\begin{aligned}
&ds\ isCycleFree \Leftrightarrow \\
&\quad \forall x, y : ds \mid x\ hasSuccessor* \ y \bullet \neg x\ hasPredecessor* \ y\ . \\
&x\ isReachableFrom\ y \Leftrightarrow x\ isSuccessor* \ y\ . \\
&x\ isShared \Leftrightarrow \\
&\quad \exists y, z : ds \bullet x\ hasPredecessor\ y \wedge x\ hasPredecessor\ z\ .
\end{aligned}
$$

where ds represents some data structure, and x and y are two objects. These invariants are challenging to verify, and are the focus of an active area of research in software verification [18][19].

5 GoF Evaluation

We used Alas to specify the GoF pattern catalog (omitting the Interpreter pattern as a domain-specific special case of the Composite pattern). Table 1 shows the invariant categories common to Alas and other DPSLs. Class diagrams with directed and aggregation associations and generalizations are ubiquitous, and are omitted for the sake of brevity. Over-riding is required in numerous GoF patterns and can be specified in the usual way in UML: by including the method or attribute in the over-riding subclass definition. Class identity involves comparing two class names for equality (names are unique) and is done using the notation `objectRole.class`, similarly to the OCL `objectRole.oclIsTypeOf( class )`. In both tables (Table 1 and 2), patterns with no relevant invariant categories are omitted, though they have been specified.

Table 2 outlines pattern invariants belonging to novel invariant categories in Alas, as well as the non-standard and clarified UML elements required for each pattern specification. It can be seen that 11 patterns can be described in more detail using the novel invariant categories, with seven benefitting from either flexible role-actor binding, inter-object state dependency or data structure. Eleven GoF pattern specifications make use of non-standard UML syntax and semantics, with object identity or value equality concepts being used in six patterns, some of which were creational, structural or behavioural. Four patterns have conditional control-flow, method sets are used to specify two patterns while flexible object role-actor bindings are used in two patterns: Observer and Composite. (Ordered) method sets are beyond the scope of this paper.

The distinction between interface and implementation dependency and data-structure invariants occur in five and three patterns respectively. Inter-object state dependency invariants occur in only two GoF patterns, but this is also an ongoing software verification challenge, with implications for modular reasoning and non-functional properties such as extensibility and maintainability [20], so they have already been shown to have widespread application outside the GoF pattern catalog. While the flexible object role-actor binding occurs only in the specification of the Observer pattern, it is useful wherever an operation is applied to every element in an unbounded (growable) collection. Finally, method

Table 1. Pattern invariants common to Alas and state-of-the-art DPSLs in GoF design pattern catalog specifications

Design pattern	Invariant type
Abstract Factory	Cardinality
Factory Method	Control-flow
Singleton	Control-flow (conditional), object state (null)
Adapter	Control-flow
Bridge	Control-flow
Composite	Object-state (Set operations)
Decorator	Interface dependency, control-flow
Façade	Interface dependency, control-flow
Flyweight	Control-flow (conditional), object state (null)
Proxy	Interface dependency, control-flow (conditional)
CoR	Control-flow (conditional)
Command	Control-flow
Iterator	Object state (Sequence operations)
Mediator	Class identity, interface dependency, control-flow
State	Class identity, control-flow
Strategy	Interface dependency, control-flow
Visitor	Cardinality, control-flow

Table 2. Novel invariant categories in Alas and non-standard UML syntax and semantics in GoF design pattern catalog specifications

Design pattern	Invariant type	Non-standard UML
Abstract Factory	Implementation dependency	Object identity
Builder	Implementation dependency	Method set
Factory Method	-	Object identity
Prototype	-	Value equality
Singleton	-	Control-flow (conditional)
Composite	Data-structure	Flexible role-actor binding
Decorator	Data-structure	-
Flyweight	-	Control-flow (cond.), object identity
Proxy	-	Control-flow (conditional)
CoR	Data-structure	Control-flow (cond.), object identity
Command	Implementation dependency	-
Iterator	Implementation dependency	Object identity
Memento	Inter-object state dependency	-
Observer	Inter-object state dependency	Flexible role-actor binding
Strategy	Implementation dependency	
Template Method	-	(Ordered) method set

sets occur in only two patterns, and ordered methods only in relation to the Template Method pattern. It is conceivable that ordered sets of methods occur frequently in software frameworks, but future work will include searching for more situations where this concept is applicable.

6 Summary, Conclusions and Future Work

In this paper, we present Alas, a design pattern specification language capable of expressing a number of invariant categories not addressed by state-of-the-art DPSLs. Each of these categories is motivated by examples from the GoF design pattern catalog and an example of an Alas specification of each category is presented. While UML is the de facto standard in object-oriented software modelling, patterns provide a different modelling challenge, as illustrated by the large body of literature on DPSLs. Also, formal software verification requires specifications with precise semantics. For this reason, non-standard UML syntax and semantics is introduced and defined. The specifications of all but one of the GoF patterns using Alas are classified according to the invariant categories they require, and the increased expressiveness of Alas with respect to the state-of-the-art is found to provide a benefit for just under half of the catalog.

In a short paper, it has not been possible to describe all the features of the language and their precise meaning. Some of the features omitted or only mentioned in this paper are Alas structural diagrams (UML Class diagrams with some modifications), cardinality invariants, pattern variant specfication, legal interleavings of pattern and non-pattern behaviour and temporal operators, including path operators (similar to LSC [21] hot and cold charts).

Planned future work includes specifying patterns outside the GoF catalog to evaluate the general applicability of Alas. Currently, the semantic definition of Alas is also incomplete. A verification tool capable of demonstrating a refinement relation between a pattern specification and a design is currently under development. Finally, as patterns impose invariants on all the members of an inheritance hierarchy, the concept of behavioural subtyping is relevant. More work is required to understand the obligations of Alas behavioural invariants on all subclasses of a specified class role.

Acknowledgements The authors would like to thank Mélanie Bouroche and Serena Fritsch for reading drafts and providing feedback. This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I3031 to Lero - The Irish Software Engineering Research Centre (www.lero.ie).

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)

2. Le Guennec, A., Sunyé, G., Jézéquel, J.: Precise Modeling of Design Patterns. In: In Proceedings of UML00, Springer Verlag (2000) 482–496
3. Group, O.M.: Object Constraint Language, Version 2.0 (2006)
4. Helm, R., Holland, I.M., Gangopadhyay, D.: Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. SIGPLAN Not. **25**(10) (1990) 169–180
5. OMG: Unified Modeling Language: Superstructure http://www.omg.org/docs/formal/09-02-02.pdf (2009)
6. Eden, A.H.: Formal Specification of Object-Oriented Design. In: Proceedings of the International Conference on Multidisciplinary Design in Engineering. (2001)
7. Mak, J.K.H., Choy, C.S.T., Lun, D.P.K.: Precise Modeling of Design Patterns in UML. In: ICSE '04, Washington, DC, USA, IEEE Computer Society (2004) 252–261
8. Lauder, A., Kent, S.: Precise Visual Specification of Design Patterns. In: ECOOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming, London, UK, Springer-Verlag (1998) 114–134
9. Flores, A., Cechich, A., Aranda, G. In: A Generic Model of Object-Oriented Patterns Specified in RSL. IGI Publishing (2007) 44–72
10. Lano, K., Bicarregui, J., Goldsack, S.: Formalising Design Patterns. In: RBCS-FACS Northern Formal Methods Workshop. (1996)
11. Taibi, T., Ngo, D.C.L.: Formal Specification of Design Patterns - A Balanced Approach. Journal of Object Technology **2**(4) (2003) 127–140
12. Dong, J., Alencar, P., Cowan, D. In: Formal Specification and Verification of Design Patterns. IGI Publishing (2007) 94–108
13. France, R.B., Kim, D.K., Ghosh, S., Song, E.: A UML-Based Pattern Specification Technique. IEEE Transactions on Software Engineering **30**(3) (2004) 193–206
14. Wendehals, L., Orso, A.: Recognizing Behavioral Patterns at Runtime using Finite Automata. In: WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis, New York, NY, USA, ACM (2006) 33–40
15. Demuth, B., Wilke, C.: Model and Object Verification by Using Dresden OCL. In: Proceedings of the Russian-German Workshop "Innovation Information Technologies: theory and practice", July 25-31, Ufa, Russia, 2009. (2009)
16. Dwyer, M., Hatcliff, J., Howell, R.: Lecture 14: Advanced OCL Expressions (2001) Kansas State University.
17. Lund, M.S., Stølen, K.: A Fully General Operational Semantics for UML 2.0 Sequence Diagrams with Potential and Mandatory Choice. In: FM 2006: Formal Methods. (2006)
18. Sagiv, M., Reps, T., Wilhelm, R.: Parametric Shape Analysis via 3-Valued Logic. ACM Transactions on Programming Languages and Systems **24**(3) (2002) 217–298
19. Berdine, J., Calcagno, C., Cook, B., Distefano, D., OHearn, P.W., Wies, T., Yang, H.: Shape Analysis for Composite Data Structures. In: Computer Aided Verification. (2007)
20. Clarke, D.G., Potter, J.M., Noble, J.: Ownership Types for Flexible Alias Protection. SIGPLAN Not. **33**(10) (1998) 48–64
21. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. In: Formal Methods in System Design, Kluwer Academic Publishers (1998) 293–312