Extending Alloy with Partial Instances

Vajih Montaghami and Derek Rayside

University of Waterloo
{vmontagh,drayside}@uwaterloo.ca

Abstract. Kodkod, the backend of Alloy4, incorporates new features for solving models where part of the solution, that is, a *partial instance*, is already known. Although Kodkod has had this functionality for some time, it is not explicitly available to the modeller through the Alloy language syntax. We propose an extension to the Alloy language to make partial instances explicitly available to the Alloy user. Explicit partial instances are helpful for the Alloy user in a number of capacities, including test-driven development, regression testing, modelling by example, and combined modelling and meta-modelling. The proposed syntax also gives the modeller explicit access to the performance benefits of Kodkod's partial instance features.

1 Introduction

Five years ago, while introducing Kodkod [9, 10], Torlak & Jackson wrote that *Alloy's main deficiency as a general-purpose problem description language is its lack of support for partial instances* [10]. (Kodkod is the backend of Alloy4.) This statement is still true for the majority of Alloy users today: despite Kodkod's support for partial instances, the Alloy language has not yet been extended to explicitly support them. In this paper, we propose a syntactic extension to the Alloy language that exposes this functionality of Kodkod. We also discuss several reasons why Alloy users might find this functionality useful. While Torlak & Jackson [10] demonstrate that Kodkod performs well on problems with partial instances, they do not describe the software engineering benefits of integrating partial instances with Alloy models.

Figure 1 introduces our syntax extension by describing three instances of a linked list: simple, single, and cyclic. In the simple instance, the line Node = head + middle + tail says that there are exactly three node atoms and their names are head, middle, and tail. The next two lines give exact bounds for the next and val relations in terms of these atoms and the integers. The single and cyclic instances are defined in a similar manner.

The inst block gives the Alloy user direct access to Kodkod's partial instance feature. Previously, if the specifier wished to specify an instance, then she would have had to do it implicitly either by constraint or by a constant function. Consider the phrase val = n→0 which gives an exact bound for the val relation in the single instance of Figure 1. In Alloy4, the specifier could have achieved a similar semantic result with the constraint fact {val = n→0} or by commenting out the

val relation declaration and introducing a constant function of the same name: fun val[] : Node→Int {n→0}. As we describe below, our new syntax extension affords the specifier greater clarity and modularity, and corresponds to a more consistently efficient translation.

Fig. 1 Alloy model of a linked list with instances expressed in proposed syntax

```
1 sig Node { next : lone Node, val : one Int }
2 inst simple { Node = head + middle + tail, -- introduce three atoms
3                next = head→middle + middle→tail, -- exact bound for next relation
4                val = head→0 + middle→1 + tail→2 } -- exact bound for val relation
5 inst single { Node = n, no next, val = n→0 }
6 inst cyclic { Node = a + b, next = a→b + b→a, val = a→0 + b→1}
```

Paper organization. Section 2 describes four ways in which partial instances benefit the Alloy user: test-driven development, regression testing, modelling by example, and combined modelling and meta-modelling. Section 3 describes our proposed extension to Alloy. Section 4 presents two experiments that demonstrate the increased computational efficiency of directly exposing Kodkod's partial instance feature when compared to adoption of traditional Alloy syntax. Section 5 considers two other possible ways to make Kodkod's partial instance feature available to Alloy users, and argues that our main proposal is preferable. Section 6 concludes.

2 Using Alloy with Partial Instances

We explore four use cases that demonstrate the utility of adding partial instances to the Alloy surface syntax: test-driven development, regression testing, modelling by example, and combined modelling and meta-modelling.

2.1 Test-Driven Development

Partial instances enables modellers to apply the test-driven development [2] methodology to their Alloy models. Consider the following example scenario. When we teach Alloy to senior undergraduates, the first in-class exercise is to write invariants for a binary tree. The lecturer, who has a computer running Alloy, displays the skeletal Alloy model listed in Figure 2.

The lecturer runs the simulation, the class looks at the result and tells the lecturer in plain language what is wrong with the displayed instance, and then the lecturer translates that plain language into formal constraints within the wellFormedTree predicate.

During this initial exercise, it is common for students to identify an instance of the model where some node y is both the left and right child of

Fig. 2 A skeletal Alloy model of a binary tree

```
1 sig Node { left, right : lone Node, val : one Int}
2 pred wellFormedTree[] { } -- to be filled in by students
3 run wellFormedTree for 3
```

some node x. When this occurs, the students usually give a constraint such as 'the left and right children cannot be equal,' which the lecturer translates as all n : Node | n.left != n.right. The students tend to be satisfied with this translation, but the astute reader will notice that this formalization prevents leaf nodes, forcing the tree to be cyclic (*i.e.*, a leaf node has no left child and no right child, and clearly the empty set is equal to the empty set). The students typically do not realize this overconstraint for fifteen or twenty minutes.

Had the students been following test-driven development with partial instances, they may have realized the folly of the proposed formalization sooner. Suppose that the students had first written the two simple partial instances in Figure 3. Figure 3a lists a tree of a single node that the students expect to be legal. Figure 3b lists a tree with self-loops that the students expect to be illegal. When the wellFormedTree predicate is empty at the beginning of the lecture the illegal self-loops test fails. When the bogus constraint n.left != n.right is added then the singleton tree test fails. Having concrete tests (partial instances) to detect errors in the program (model) is the essence of test-driven development.

Fig. 3 Two partial instances of a binary tree: (a) a legal singleton tree, and (b) a tree with illegal self-loops

(a)
```
1 inst SingletonTree { Node = n, no left, no right, val = n→0 }
2 run wellFormedTree for SingletonTree expect 1
```

(b)
```
1 inst IllegalSelfLoops { Node = n, left = n→n, right = n→n }
2 run wellFormedTree for IllegalSelfLoops expect 0
```

A difference between test-driven development for imperative code versus that for declarative logic models is the role of positive and negative examples. With imperative code, the programmer writes positive test cases for empty procedures (or code stubs) that initially fails. With declarative logic models, a positive example (such as a singleton tree) will succeeds with an empty wellFormedTree predicate. Only once the predicate becomes overconstrained will the positive example fail. In contrast, negative examples will fail with the empty predicate, and will only pass with a properly constrained predicate. Consider, for example, the negative example of a node that is its own child in Figure 3b. If wellFormedTree is underconstrained (*e.g.*, empty) then this test will fail. Thus, the programmer

builds up a procedure to construct positive examples, the modeller builds up a predicate to rule out negative examples.

2.2 Regression Testing of Alloy Models

Like programs, specifications evolve: requirements change, extra properties need to be checked, refactoring for readability, and so on. As with programs, some form of regression testing can provide assurance that the specification (or program) still corresponds to programmer intent.

For an Alloy specification with associated safety properties, partial instances can be used in regression testing to detect over-constrained models. When a model becomes over-constrained, the safety properties will still hold; however, the modeller might be unaware of over-constraints. Regression testing of Alloy instances can be effective in detecting these occurrences.

The user following a TDD approach can have their initial tests do double duty as regression tests.

2.3 Modelling by Example

The idea of modelling by example [7] is that the system induces logical constraints through a dialogue of examples with the user. The user begins by providing some prototypical instances to the system, and then the system responds with other instances that the user classifies as either valid or invalid. As the dialogue continues the system refines a general formula that includes the positive examples and excludes the negative examples.

A modelling by example system would be substantially facilitated by having explicit syntactic support for partial instances in Alloy.

2.4 Combined Modelling and Meta-Modelling

Alloy is sometimes used to define new modelling languages. We will refer to such activity as 'meta-modelling.' Let L name the Alloy model that describes the new language, and let M name an Alloy model that describes a model written in the new language. At present, there is often no mechanical connection between L and M. Our facility for adding partial instances to Alloy makes it easier to have L and M tightly integrated. We examine the work of Cai & Sullivan *et alia* as a case study to illustrate these points.

In a series of papers over the last ten years Cai & Sullivan *et alia* have been exploring formal techniques for assessing modularity in software design [3–5, 8]. This is a serious, high-quality research effort that (we claim) illustrates some of the shortcomings of the current Alloy surface syntax that our proposal for integrating partial instances addresses.

Cai & Sullivan have written their meta-model (L) in Z [3]. This meta-model is then implicitly encoded in the Java source of their tool Simon. Given a model of a software design in their language, Simon produces a specialized Alloy model

Fig. 4 Partial instance encoding of Irwin *et alia*'s description [6] of the design space for a matrix manipulation program

```
1  inst IrwinMatrixDesignSpace {
2    AugmentedConstraintNetwork = ACN,
3    Variable = Density + Struct + Alg,
4    Value = dense+sparse + links+array + traverse+lookup + other,
5    domain = Density→(dense+sparse) + Struct→(links+array+other) +
6      Alg→(lookup+traverse+other),
7    dominates = ACN →((Struct→Density)+(Alg→Density)),
8    solutions = ACN →Solution,
9  }{−−Appended facts have access to atom names introduced in inst block
10   all s : Solution | {
11     let x = {p : Variable, q : Value | some b : s.bindings | p=b.var and q=b.val} |{
12       (Struct→links) in x ⇒ (Density→sparse) in x
13       (Struct→array) in x ⇒ (Density→dense) in x
14       (Alg→lookup) in x ⇒ (Struct→array) in x
15       (Alg→traverse) in x ⇒ (Struct→links) in x
16   }}
17 }
18 run createMatrixACN for IrwinMatrixDesignSpace
```

(M) that is used to check modularity properties of the proposed software design. There is no mechanically analyzed connection between L and M.

We have translated the Cai & Sullivan meta-model from Z to Alloy and used our partial instance feature to write some of Cai & Sullivan's specific models as partial instances of this meta-model. Figure 4 lists our encoding of Cai & Sullivan's study of Irwin *et alia*'s example of designing a program to store and manipulate a matrix [6]. There are three variables (decisions) in this design space (line 3): the density of the matrix, the underlying data structure used to encode the matrix, and the algorithm used to manipulate that structure. More specifically, the matrix may be dense or sparse, the structure may be a linked list or an array (or other), and the algorithm may be either 'lookup' or 'traversal' (or other) (lines 4–6). In the vocabulary of Cai & Sullivan, the density decision dominates the data structure and algorithm decisions (line 7). The intuition here is that one selects the data structure and algorithm depending on whether the density is expected to be dense or sparse. Additionally, the partial instance block is followed by a list of facts (lines 9–17) that constrain valid solutions of the design space to those where the algorithm and data structure are natural matches for the matrix density and each other. A fact appended to a partial instance block can make use of the atom names introduced in that block.

We now have a mechanically analyzed connection between the Cai & Sullivan meta-model and the specific model of the design space of a matrix manipulation program. We have greater certainty that the properties we have checked on the meta-model also hold of the model (partial instance).

Clafer [1] is a language that is designed to support combined modelling and meta-modelling.

3 Language Extension

We propose to add an inst block to the Alloy language, allowing the user to specify a partial instance, as illustrated above in Figures 1, 2, 3, and 4. The partial instances in those examples only use exact bounds; Kodkod and our syntax also support lower and upper bounds as well, using the in and includes keywords, respectively. Lower bound is a set of tuples that a relation must have, and upper bound is the one that realation might have [9].

These inst blocks are given names and used in Alloy commands. Whereas now a user might write run p for 3, they will now write run p for i, indicating that predicate p is to be simulated in the context of partial instance i.

An inst block, like a sig block, may have an appended fact. For inst blocks, the appended fact is only expected to be true when that inst block is part of the command being executed. The purpose of this appended fact is to give the specifier an opportunity to write constraints that mention the atom names introduced in the inst block — these names are not available elsewhere in the model.

Fig. 5 Grammar and preliminary type definitions

(a) Grammar

⟨*iBlk*⟩ := 'inst' id ('extends' id)? '{' ⟨*iSt*⟩[,⟨*iSt*⟩]* '}' ('{' ⟨*frml*⟩ '}')?

⟨*iSt*⟩ := ⟨*n*⟩
| 'exactly' ⟨*n*⟩ ⟨*var*⟩
| ⟨*var*⟩ '=' ⟨*iXpr*⟩
| ⟨*var*⟩ 'in' ⟨*iXpr*⟩
| ⟨*var*⟩ 'include' ⟨*iXpr*⟩
| ⟨*var*⟩ 'include' ⟨*iXpr*⟩ 'moreover' ⟨*iXpr*⟩
| 'no' ⟨*var*⟩

⟨*iXpr*⟩ := ⟨*iXpr*⟩ '->' ⟨*iXpr*⟩
| ⟨*iXpr*⟩ '+' ⟨*iXpr*⟩
| '(' ⟨*iXpr*⟩ ')'
| ⟨*atm*⟩

(b) Preliminary type definitions

⟨*prb*⟩ := ⟨*univ*⟩ ⟨*iSt*⟩* ⟨*frml*⟩*

⟨*univ*⟩ := {⟨*atm*⟩[,⟨*atm*⟩]*}

⟨*tpl*⟩ := ⟨*atm*[,*atm*]*⟩

⟨*cnst*⟩ := {tpl[,tpl]*} | {} [×{}]*

⟨*var*⟩ := id

⟨*atm*⟩ := id

⟨*sig*⟩ := ⟨*var*⟩

⟨*sigs*⟩ := ⟨*sig*⟩*

⟨*n*⟩ := int

Figure 5a lists the grammar for our proposed extension to the Alloy language to support partial instances. An iBk has a name, a list of iSts and optionally an appended fact. Each iSt alternative that contains a var bounds either a signature or a field (whichever is named by the var). The one iSt alternative that does not name a var provides the default number of atoms for each signature. A relation (signature or field) name can only appear on the left-hand side of at most one iSt in each iBk.

An iSt that names a signature on its left-hand side introduces atom names on its right-hand side. These atom names can then be used to describe the bounds on fields. An iXpr is an expression that describes a set of tuples using the normal

Alloy union (+) and cross-product (→) operators along with the names of the atoms. If the user wishes to specify both an upper and lower bound for relation r, they can write an iSt like r include x + y moreover p + q, which specifies a lower bound of x + y and an upper bound of x + y + p + q.

One partial instance block may extend another. For example, the partial instance in Figure 10 extends the partial instance in Figure 4. The semantics of partial instance extension are simply concatenation and conjunction. Let p name the base partial instance block; let q name the extending partial instance block; and let r name the result of applying the extension to q. The text of r is the concatenation of the text of p with the text of q. The appended fact of r is the conjunction of p's appended fact with q's appended fact. The result r must follow the same well-formedness guidelines as p and q: no relation can be named on the left-hand side of more than one statement. This restriction keeps both regular semantics and extension semantics simple, as it prevents statements from interfering with each other (notwithstanding quantitative statements that interact with named statements in a well-defined manner as formalized below).

Fig. 6 Universe construction

$evr : sig \to univ$
$U : iBlk \to sigs \to evr$
$G : iSt^* \to sigs \to evr \to univ \quad G' : iSt \to sigs \to evr$
$X : iSt^* \to sigs \to evr \to evr \quad X' : iSt \to sigs \to evr \to evr$
$N : iSt^* \to sigs \to evr \quad N' : iSt \to sigs \to evr$
$K : sig \to int \to univ$
$Q : iXpr \to univ$

$$
\begin{aligned}
&U[\![iBlk, sigs]\!] && := G[\![iSt_1 \ldots iSt_n, sigs, X[\![iSt^*, sigs, N[\![iSt^*, sigs, \emptyset]\!]]\!]]\!] \\
&G[\![iSt^*, sig, evr]\!] && := G[\![iSt_1 \ldots iSt_n, sigs, evr]\!] \\
&G[\![iSt_1 \ldots iSt_n, sigs, evr]\!] && := G[\![iSt_2 \ldots iSt_n, sigs, evr \ ++ \ G'[\![iSt_1, sigs]\!]]\!] \\
&G[\![[], sigs, evr]\!] && := evr \\
&G'[\![v \ [=|\textbf{in}|\textbf{include}] \ p, sigs]\!] && := \{(a,b) | a \in sigs \wedge a = v \wedge b \in Q[\![p]\!]\} \\
&G'[\![v \ \textbf{include} \ p \ \textbf{moreover} \ q, sigs]\!] && := \{(a,b) | a \in sigs \wedge a = v \wedge b \in Q[\![p]\!] \cup Q[\![q]\!]\} \\
&X[\![iSt^*, sigs, evr]\!] && := X[\![iSt_1 \ldots iSt_n, sigs, evr]\!] \\
&X[\![iSt_1 \ldots iSt_n, sigs, evr]\!] && := X[\![iSt_2 \ldots iSt_n, sigs, evr \ ++ \ X'[\![iSt_1, sigs]\!]]\!] \\
&X[\![[], sigs, evr]\!] && := evr \\
&X'[\![\textbf{exactly} \ n \ v, sigs]\!] && := \{(a,b) | a \in sigs \wedge \ a = v \wedge b \in K[\![v, n]\!]\} \\
&N[\![iSt^*, sigs, evr]\!] && := N[\![iSt_1 \ldots iSt_n, sigs, evr]\!] \\
&N[\![iSt_1 \ldots iSt_n, sigs, evr]\!] && := N[\![iSt_2 \ldots iSt_n, sigs, evr \ ++ \ N'[\![iSt_1, sigs]\!]]\!] \\
&N[\![[], sigs, evr]\!] && := evr \\
&N'[\![n, sigs]\!] && := \{(a,b) | a \in sigs \wedge b \in K[\![a, n]\!]\} \\
&K[\![v, n]\!] && := \{\langle ToString(v) +' \ \$' + ToString(n-1)\rangle\} \cup K[\![v, n-1]\!] \\
&K[\![v, 0]\!] && := \langle\rangle \\
&Q[\![p]\!] && := \{\langle ToString(p)\rangle\} \\
&Q[\![p + q]\!] && := Q[\![p]\!] \cup Q[\![q]\!] \\
&Q[\![p \to q]\!] && := \langle\rangle
\end{aligned}
$$

3.1 Semantics

We define the semantics of the partial instance block as an extension of the Kodkod semantics [9]. The Kodkod semantics take a universe and relation bounds as inputs. The purpose of the partial instance block is for the user to specify the universe and relation bounds.

Figure 6 describes how the universe is constructed from a partial instance block by the U function, which in turn makes use of the N, X, and G functions. Preliminary type definitions are given above in Figure 5b. First the N function constructs a universe in which each sig has the default number of atoms. The X function takes this default universe and returns a universe that complies with the exactly statements in the partial instance block. Finally, the G function adds atoms named in upper and lower bound statements. All of these functions take as input a set of the sigs declared in the model. This set of sig names is used to distinguish statements that might introduce atoms (which name a sig on the left-hand side) from statements that bound relations (which name a field on the left-hand side).

Once the universe is constructed (Figure 6), then the bounds can be constructed (Figure 7). Figure 7 starts by redefining the top-level function P from the Kodkod semantics [9] to indicate that the universe and the relation bounds are generated from the partial instance block.

Fig. 7 Bounds construction (building on formalization of [9])

$P : problem \to binding \to boolean$		— top-level function, re-defined from [9]
$F : formula \to binding \to boolean$		— formulas, definition given in [9]
$S : iSt^* \to sigs \to evr \to binding \to boolean$		— list of inst statements
$S' : iSt \to sigs \to evr \to binding \to boolean$		— individual inst statement
$C : iXpr \to univ \to cnst$		— expressions
$W : var \to sigs \to evr \to univ$		—
$P[\![sigs.U[\![iBk, sigs]\!]\ iSt_1 \ldots iSt_n\ \ frml^*]\!]_b$	$:= S[\![iSt_1 \ldots iSt_n, sigs, U[\![iBk, sigs]\!]]\!]_b \wedge F[\![frml^*]\!]_b$	
$S[\![iSt_1 \ldots iSt_n, sigs, evr]\!]_b$	$:= S[\![iSt_2 \ldots iSt_n, evr, sigs]\!]_b \wedge S'[\![iSt_1, evr, sigs]\!]_b$	
$S[\![[], evr, sigs]\!]_b$	$:= true$	
$S'[\![\textbf{exactly}\ n\ v, evr, sigs]\!]_b$	$:= W[\![v, sigs, evr]\!] \subseteq b(v) \subseteq W[\![v, sigs, evr]\!]$	
$S'[\![v{=}p, evr, sigs]\!]_b$	$:= C[\![p, sigs.evr]\!] \subseteq b(v) \subseteq C[\![p, sigs.evr]\!]$	
$S'[\![v\ \textbf{in}\ p, evr, sigs]\!]_b$	$:= C[\![\emptyset, sigs.evr]\!] \subseteq b(v) \subseteq C[\![p, sigs.evr]\!]$	
$S'[\![v\ \textbf{include}\ p, evr, sigs]\!]_b$	$:= C[\![p, sigs.evr]\!] \subseteq b(v) \subseteq W[\![v, sigs, evr]\!]$	
$S'[\![v\ \textbf{include}\ p\ \textbf{moreover}\ q, evr, sigs]\!]_b$	$:= C[\![p, sigs.evr]\!] \subseteq b(v) \subseteq C[\![p + q, sigs.evr]\!]$	
$S'[\![\textbf{no}\ v]\!]_b$	$:= b(v) = \emptyset$	
$C[\![p + q, univ]\!]$	$:= C[\![p, univ]\!] \cup C[\![q, univ]\!]$	
$C[\![p \to q, univ]\!]$	$:= \{\langle p_1, \ldots, p_n, q_1, \ldots, q_m\rangle \mid \langle p_1, \ldots, p_n\rangle \in C[\![p, univ]\!] \wedge \langle q_1, \ldots, q_m\rangle \in C[\![q, univ]\!]\}$	
$C[\![p, univ]\!]$	$:= \{p' \mid p' \in univ\ \wedge\ ToString(p') = p\}$	
$W[\![v, sigs, evr]\!]$	$:= \{\langle p_1, \ldots, p_n\rangle \mid (v \in sigs\ \implies\ p_1 \in v.evr) \wedge (v \notin sigs\ \implies\ p_i \in v_i.evr)\}$	

4 Experiments

We performed two experiments to evaluate the computational efficiency of the proposed partial instance block: a micro-benchmark to characterize the maximum possible improvement, and our combined modelling and meta-modelling case-study based on Cai & Sullivan's work (§2.4). All tests are done on Intel i7-2600K CPU at 3.40GHz with 16GB memory. The performance results are essentially the same with both Minisat and Sat4J, although we report only the Sat4J results here.

We compared using the partial instance block to two alternative specification styles in two different versions of Alloy 4.2. The two different styles were constraining relations with facts and using constant functions instead of relations.

Constant functions are just expressions that are inlined at their point of use. They add clauses but not variables to the generated SAT formula. Alloy 4.x includes some inference capability to translate constraints on relations as bounds. In response to a draft of this paper the Alloy development team improved this inference capability. We refer to this enhanced version as A4.2′, and to the version of Alloy 4.2 from January 2012 as A4.2. We refer to our version of Alloy with the partial instance block as A4.2*i*.

4.1 Micro Benchmark

We devised a micro-benchmark to illustrate the upper bound on the potential performance improvements of exposing Kodkod's partial instance features through our new syntax. Our micro-benchmark has a single signature S and a single binary relation r that maps S to S. For our partial instance, we want to introduce some named atoms of sig S, and then define relation r to be a fully connected graph (*i.e.*, map every S atom to every other S atom).

Figure 8 lists examples of these partial instance models in the three different syntaxes: *(a)* constraining relation r with a fact; *(b)* replacing relation r with a constant function named r; and *(c)* using our new partial instance syntax. The example listings in Figure 8 show these models where signature S has two atoms (S0 and S1). For the plots in Figure 9, we generated these models with signature S having up to seventy-five atoms. The cardinality of relation r is proportional to the square of the cardinality of signature S (as one would expect from a fully connected graph).

Fig. 8 Example models for micro-benchmark experiment

(a) By Fact	*(b)* By Constant-Function	*(c)* By Inst-Block
one sig S0,S1 **extends** S{} **fact** {r=S0→S1 + S1→S0} **pred** f[]{**all** s:S | S **in** s.^r} **run** f	**one sig** S0,S1 **extends** S{} **fun** r[]:S→S{S0→S1 + S1→S0} **pred** f[]{**all** s:S | S **in** s.^r} **run** f	**inst** b { S=S0 + S1, r=S0→S1 + S1→S0} **pred** f[]{**all** s:S | S **in** s.^r} **run** f **for** b

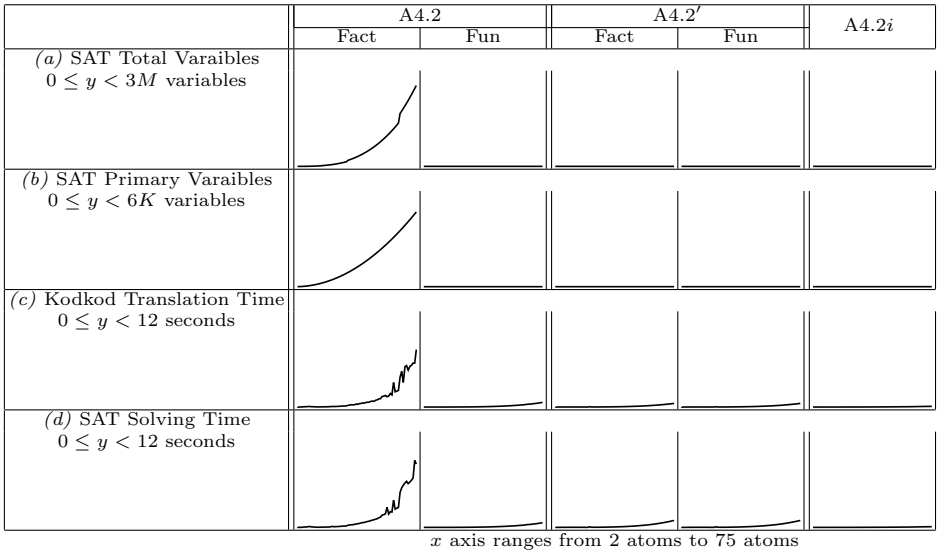
Figure 9 shows graphs characterizing how the translations of the three syntactic approaches shown in Figure 8 scale on different measures: *(a)* total number of variables in the resulting boolean (SAT) formula; *(b)* number of primary variables in the resulting boolean (SAT) formula; *(c)* time taken by Kodkod to translate the Alloy model to SAT; and *(d)* time taken by the SAT solver to find a solution. We make a number of observations from the data in Figure 9:

1. The inference capability of A4.2 is incomplete: it is unable to deduce that the constraints on r can be translated as bounds rather than as variables and clauses. Therefore the number of variables and the translation and solving times grow exponentially.
2. All other strategies show very little growth as the number of atoms increases.

3. The improved inference in A4.2′ is effective (A4.2′ Fact column).
4. The number of SAT variables produced by the constant function encoding, the improved inference, and the partial instance strategies is the same (low).
5. The partial instance encoding has the fastest translation and solving times (by a narrow margin).

The main conclusion of Figure 9 is that if the specifier chooses to use constant functions instead of relations or writes their facts in a manner that Alloy can infer bounds from, then there is little performance gain from the partial instance block. However, the partial instance block does provide the best performance, and does so without the specifier having to worry about whether their writing style is comprehensible to Alloy's bounds inference facility.

Fig. 9 Results of micro-benchmarks.



4.2 Staged Evaluation

The proposed partial instance feature offers Alloy users the opportunity to stage evaluation of their models, which might potentially save time when certain parts of the model are not changing and other parts are. Consider, for example, the model in Figure 4 that describes the design space of a program to manipulate matrices. The partial instance of Figure 4 is written in terms of Cai & Sullivan's meta-model, which has (design) variables, values, bindings of variables to values, and 'states'. (The 'states' are of a design automaton, which is a concept they use to analyze design spaces that we do not explain here.)

Suppose that the user wishes to experiment with the constraints written in the appended fact of Figure 4. These constraints do not affect the space of valid binding atoms. Therefore, the user could stage the evaluation of the model by saving the legal bindings in a partial instance, such as in Figure 10. Subsequent simulations would not have to re-solve this part of the model.

Fig. 10 Iwrin matrix design space partial instance (Figure 4) extended with binding atoms generated by a previous simulation

```
1 inst IrwinMatrixDesignSpace_WithBindings extends IrwinMatrixDesignSpace {
2   Binding = B0+B1+B2+B3+B4+B5+B6+B7,
3   var = (B0+B1)→Struct + (B2+B3+B4)→Density + (B5+B6+B7)→Alg,
4   val = B0→dense + B1→sparse + B2→links + B3→array + B4→other +
5         B5→traverse + B6→lookup + B7→other }
6 run createMatrixACN for IrwinMatrixDesignSpace_WithBindings
```

Figure 11 characterizes the potential performance improvements from staged evaluation using the Irwin matrix design space example of Cai & Sullivan. The translation time for the model from Figure 10 is over ten times faster than the translation time for the model from Figure 4, and the solving time is three times faster, for an overall improvement of seven times. Obviously the speedup to be gained from staged evaluation depends on the particulars of the model in question; other models will likely produce different results than this one.

Figure 11 also shows performance results for A4.2 and A4.2′ simulating a model equivalent to Figure 4 (*i.e.*, not staged). In this particular case there is no significant difference between A4.2 and A4.2′. We suspect that this is the case because the domain relation is constrained piecewise across a number of appended facts. All of these piecewise constraints add up to an exact bound on domain, but a fairly sophisticated whole-model analysis would be needed to deduce that. A4.2*i* results in four times faster solving time than A4.2′ for the model in Figure 4, at the expense of a 10% slowdown in translation time.

Fig. 11 Performance improvements from staged evaluation

	Total Vars	Pri. Vars	Clauses	Translation time (ms)	Solving time (ms)
A4.2*i* (Fig. 4)	59,694	773	162,642	12,742	6,744
A4.2*i* (Fig. 10 — staged)	20,060	503	37,148	986	2,174
A4.2	59,953	768	162,417	11,976	27,415
A4.2′	59,953	768	162,417	11,188	27,730

5 Alternatives Considered

In this section we consider some alternative approaches for specifying partial instances in Alloy and argue for the approach proposed in this paper.

5.1 Static Analysis

Alloy 4.x already includes the capability to infer when constraints might be encoded as Kodkod bounds rather than as SAT clauses. Although it is not yet perfect, this capability will continue to improve. Given this capability, no extra syntax is needed to realize the main performance benefits of Kodkod's partial instance feature.

We argue that there are software engineering benefits to our new syntax beyond the performance gains that it affords. The proposed syntax makes it easy for the specifier to run different commands with different instances, or to run commands with no partial instance (the norm in Alloy now). Writing a partial instance implicitly via constraints in the traditional Alloy syntax makes it difficult to switch from running a command with a partial instance to running a command without a partial instance. For example, to run the fragment in Figure 8a without a partial instance, we would want to remove the keyword abstract from the signature S and remove the sub-signatures S1, S2, S3. With the partial instance block syntax, one does not have to edit the text of the model to run it in these different ways. A number of our use cases described above depend on this affordance of the new syntax.

5.2 Syntactic Alternatives for the Partial Instance Block

There are a variety of different ways in which one could specify the body of a partial instance block. We consider the proposal described above to be a 'relational' style because each statement specifies a different relation.

Alternatively, one could imagine an 'object-oriented' syntax in which relations are defined piecewise with respect to individual atoms. Figure 12a lists a small example of this syntax. The same example is listed in the relational style in Figure 12c. The object-oriented style syntax is intuitively appealing for some examples; however, its piecewise nature makes the bound being defined unclear: does Figure 12a define a lower bound or an exact bound for relation r?

Another alternative syntax is 'set-oriented' style, shown in Figure 12b. This style is concise and consistent with common mathematical notation, but it does not conform to the existing Alloy expression grammar.

Our proposed relational style syntax (Figure 12c) conforms to the existing Alloy expression grammar and has a clear and uniform way to specify lower, exact, and upper bounds.

Fig. 12 Syntactic alternatives for the body of the partial instance block

(a) object-oriented style

```
sig S{r: S}
inst i{S=S1+S2+S3,
   S1.r=S2, S2.r=S3}
```

(b) set-oriented style

```
sig S{r: S}
inst i{S={S1,S2,S3},
       r={S1→S2,S2→S3}}
```

(c) relational style

```
sig S{r: S}
inst i{S=S1+S2+S3,
       r=S1→S2+S2→S3}
```

6 Conclusion

Explicit partial instances could be used in Alloy to efficiently specify constraints on allowable solutions (their intended usage in Kodkod); for test-driven development of Alloy models; for regression testing of Alloy models; to support new ideas such as modelling by example; and for combined modelling and meta-modelling. While Alloy currently has an inference mechanism that makes use of Kodkod's partial instance functionality behind the scenes, these engineering benefits are substantially facilitated by explicit syntactic support for partial instances.

There is more than one possible way to expose Kodkod's partial instance feature to the Alloy user. We have explored a number of alternatives and recommend a new named block with statements written in a relational style. This recommendation is backwards compatible with existing Alloy models and the existing Alloy expression grammar; it affords the user a uniform way to express exact, upper, and lower bounds; it combines with Alloy commands in a modular fashion; and it has an easy and efficient translation to Kodkod.

Kodkod has supported partial instances for five years, and that is one of its main improvements over the backend of Alloy3. It's time that Alloy users had the opportunity to take full advantage of this functionality.

Acknowledgments

We thank Daniel Jackson, Aleksandar Milicevic, Steven Stewart, Emina Torlak, and the anonymous referees for comments on previous drafts of this paper. We also thank Aleksandar Milicevic for his assistance with the Alloy code base and for improving the inference capabilities of the Alloy analyzer.

This work was supported in part by the National Science and Engineering Research Council of Canada (NSERC).

Bibliography

[1] Bak, K., Czarnecki, K., Wasowski, A.: Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled. In: Proc. of the 3rd International Conference on Software Language Engineering (SLE) (2010)

[2] Beck, K.: Test-Driven Development. Addison-Wesley, Reading, Mass. (2003)

[3] Cai, Y.: Modularity in Design: Formal Modeling and Automated Analysis. Ph.D. thesis, University of Virginia (Aug 2006)

[4] Cai, Y., Huynh, S., Xie, T.: A framework and tool supports for testing modularity of software design. In: Egyed, A., Fischer, B. (eds.) Proc.22nd ASE. pp. 441–444. Atlanta, GA (Nov 2007)

[5] Cai, Y., Sullivan, K.: Modularity analysis of logical design models. In: Easterbrook, S., Uchitel, S. (eds.) Proc.21st ASE. Tokyo, Japan (Sep 2006)

[6] Irwin, J., Loingtier, J.M., Gilbert, J.R., Kiczales, G., Lamping, J., Mendhekar, A., Shpeisman, T.: Aspect-oriented programming of sparse matrix code. In: ISCOPE. pp. 249–256 (1997)

[7] Mendel, L.: Modeling by Example. Master's thesis, MIT (Sep 2007)

[8] Sullivan, K.J., Griswold, W.G., Cai, Y., Hallen, B.: The structure and value of modularity in software design. In: Proc.9th FSE. pp. 99–108. Vienna, Austria (Sep 2001)

[9] Torlak, E.: A Constraint Solver for Software Engineering: Finding Models and Cores of Large Relational Specifications. Ph.D. thesis, MIT (2009)

[10] Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) Proc.13th TACAS. LNCS, vol. 4424, pp. 632–647. Springer-Verlag, Braga, Portugal (Mar 2007)