Under consideration for publication in Math. Struct. in Comp. Science

Applications and Extensions of Alloy: Past, Present, and Future

EMINA TORLAK[1], MANA TAGHDIRI[2], GREG DENNIS[3] and JOSEPH P. NEAR[4]

[1] *LogicBlox, Atlanta, GA, USA.*
[2] *Karlsruhe Institute of Technology, Karlsruhe, Germany.*
[3] *Google, Cambridge, MA, USA.*
[4] *Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, MA, USA.*

Received March 2011

Alloy is a declarative language for lightweight modeling and analysis of software. The core of the language is based on first-order relational logic, which offers an attractive balance between analyzability and expressiveness. The logic is expressive enough to capture the intricacies of real systems, yet it is also simple enough to support fully automated analysis with the Alloy Analyzer. The Analyzer is built on a SAT-based constraint solver and provides automated simulation, checking, and debugging of Alloy specifications. Because of its automated analysis and expressive logic, Alloy has been applied in a wide variety of domains. These applications have motivated a number of extensions both to the Alloy language and to its SAT-based analysis. The article provides an overview of Alloy in the context of its three largest application domains (lightweight modeling, bounded code verification, and test-case generation) and three recent application-driven extensions (an imperative extension to the language, a compiler to executable code, and a proof-capable analyzer based on SMT).

1. Introduction

Alloy (Jackson, 2006) is a declarative language for lightweight modeling and analysis of software systems. The core of the language is based on *relational logic*—a simple but powerful combination of first-order logic, relational algebra, and transitive closure. By design, Alloy offers an attractive balance between analyzability and expressiveness. Its underlying logic is expressive enough to capture the intricacies of real systems (*e.g.*, the flash file system (Kang and Jackson, 2009)), yet it is also simple enough to support fully automated analysis.

Alloy is equipped with three analyses, provided by the Alloy Analyzer (Chang, 2007): simulation, checking, and debugging. The Analyzer can *simulate* a system by exhaustively searching for a finite instance of its specification; it can *check* that the system has a desired property by searching for a counterexample to that property; and, finally, it can help *debug* an overconstrained specification by highlighting the constraints that conflict

with one another. All three analyses are performed by reducing Alloy to relational logic, and solving the resulting constraints with Kodkod (Torlak, 2009), an efficient, SAT-based solver for relational satisfiability problems. Kodkod works by translating a relational problem to a set of equisatisfiable boolean clauses, which are decided by an off-the-shelf SAT solver. The SAT solver is pluggable, ensuring that both Kodkod and Alloy users automatically benefit from the continuing advances in SAT solving technology.

The versatility of Alloy, coupled with the fully automated analysis, has motivated its use in a wide range of applications, both practical and exploratory. Examples include design modeling and analysis (Kang and Jackson, 2009; Ramananandro, 2008); bounded program verification (Dennis et al., 2006; Dennis, 2009; Dolby et al., 2007; Galeotti et al., 2010; Taghdiri and Jackson, 2007; Torlak et al., 2010); test-case generation (Abdul Khalek and Khurshid, 2010; Shao et al., 2007; de la Riva et al., 2010; Uzuncaova and Khurshid, 2008; Uzuncaova et al., 2008); specification extraction (Taghdiri et al., 2006); counterexample generation (Blanchette and Nipkow, 2009; Spiridonov and Khurshid, 2007); and declarative configuration (Narain et al., 2008; Yeung, 2006).

The use of Alloy by the wider community has driven a number of extensions to the language, as well as a number of alternatives to its SAT-based analysis. Each of these addresses a key challenge that emerged from the efforts to apply Alloy: (1) a lack of built-in support (such as control constructs) for modeling dynamic systems, (2) a lack of automated assistance for compiling an Alloy specification into an executable implementation, (3) a lack of support for verifying rather than just checking properties of specifications, and (4) limited support for numerical constraints. DynAlloy (Frias et al., 2005) and Imperative Alloy (Near and Jackson, 2010) tackle the first challenge by extending Alloy with imperative constructs, which enable concise modeling of dynamic systems. Several approaches have also been developed to help with the implementation and verification challenges. These include a recent compiler from Imperative Alloy to Prolog (Near, 2010); a translator from a stylized subset of Alloy to SQL (Krishnamurthi et al., 2008); an automated Alloy prover based on SMT (El-Ghazi and Taghdiri, 2011); and two interactive provers based on PVS (Frias et al., 2007) and Athena (Arkoudas et al., 2003). The SMT-based approach addresses the fourth challenge as well, by providing effective support for numerical constraints.

This article provides an overview of Alloy in the context of past and present applications, with an eye toward the future of the language and its analysis. We begin by briefly introducing the Alloy language and the Analyzer (Section 2). We then present three sets of applications, highlighting the strengths and limitations of the Alloy approach (Section 3). To conclude, we review recent efforts to address these limitations and discuss directions for future development (Sections 4-5).

2. A Brief Guide to Alloy: Relations, Models, and Cores

Alloy can be viewed roughly as a minimalist subset of Z (Spivey, 1992), with a strong connection to data modeling languages such as ER (Chen, 1976) and SDM (Hammer and McLeod, 1978). The logic of Alloy is based on a single concept—that of a *relation*. Functions are treated as binary relations; sets are unary relations; and scalars are single-

ton unary relations. An Alloy *specification* is a collection of first-order constraints over relational variables. The constituent tuples of these variables are drawn from a *universe* of uninterpreted elements, or *atoms*.

For example, the following is a tiny, stand-alone specification of LISP-style lists:

```
1  sig Thing {}
2  sig List {
3     car: lone Thing,
4     cdr: lone List
5  }
```

The specification is defined over two sets, Thing and List, and two binary relations, car and cdr. The set Thing models the objects that can be stored in Lists. The relations car and cdr model the 'car' and 'cdr' pointers of a list: car maps each List to a Thing (if any) that is stored in the list, and cdr maps every List to its tail (if any), which is also a List. The keyword **lone** constrains the pointer relations to be partial functions from their domain to their range, allowing some lists to be empty (that is, have no 'car' or 'cdr').

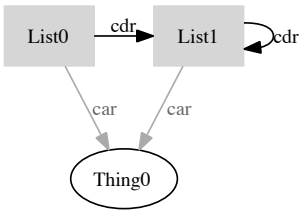
Given the above five lines of Alloy, we can use the Alloy Analyzer to *simulate* some lists, to ensure that the specification is consistent:

```
6  run {} for 3
```

The **run** command instructs the Analyzer to search for a *model*, or an *instance*, of the list specification that contains up to three lists and up to three things. An instance of a specification is a binding of its free variables—in this case, Thing, List, car, and cdr—to sets of tuples that makes the specification true.

Executing the command produces the following instance:

Thing	↦	{⟨Thing0⟩}
List	↦	{⟨List0⟩, ⟨List1⟩}
car	↦	{⟨List0, Thing0⟩, ⟨List1, Thing0⟩}
cdr	↦	{⟨List0, List1⟩, ⟨List1, List1⟩}



The Analyzer automatically produces a visualization of the instance, as shown on the right. The actual variable bindings are shown on the left for completeness. It is easy to see that the bindings satisfy the list constraints.

While demonstrating the consistency of our specification, the above instance also demonstrates that its constraints are quite weak. In particular, they allow lists with infinite (cyclic) 'cdr' pointers. To ensure that all lists terminate, we can refine the specification by introducing a special 'nil' list:

```
 7  one sig Nil extends List {}
 8  fact {
 9     no Nil.car
10     no Nil.cdr
11     all l: List − Nil | some l.car and some l.cdr
12     all l: List | Nil in l.^cdr
13  }
```

The Nil list is unique and has neither a 'car' nor a 'cdr' (lines 7-10). Non-nil lists have both (line 11), and the transitive closure (^) of every list's cdr includes Nil (line 12). That is, following the 'cdr' pointer from any list eventually leads to the 'nil' list. Because 'nil' itself has no 'cdr,' the above constraints imply that all lists are finite.

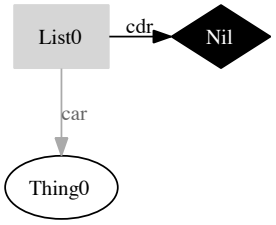
We can now re-execute the **run** command (line 6) to search for some well-formed lists. This time, however, the Analyzer finds no satisfying instance. Instead, it reports that the specification is *unsatisfiable* and produces the following *minimal unsatisfiable core*:

```
10  no Nil.cdr
12  all l: List | Nil in l.^cdr
```

A minimal core of an unsatisfiable specification is a subset of its constraints that is also unsatisfiable, but that becomes satisfiable if any of the included constraints are removed. In other words, a minimal core pinpoints an irreducible source of inconsistency in the specification. In our case, the conflicting constraints are pointing out that Nil has no cdr and, as a result, Nil cannot be reached from itself by traversing cdr one or more times.

To fix the inconsistency, we weaken the constraint on line 12 by replacing the transitive closure operator with '*', which stands for reflexive transitive closure. Since Nil can be reached from itself by traversing cdr *zero* or more times, the amended specification is satisfiable. The Analyzer confirms this by producing a satisfying instance:

Thing	↦	{⟨Thing0⟩}
List	↦	{⟨List0⟩, ⟨Nil⟩}
Nil	↦	{⟨Nil⟩}
car	↦	{⟨List0, Thing0⟩}
cdr	↦	{⟨List0, Nil⟩}



In addition to simulation and debugging, the Analyzer can also be used for *checking* whether an Alloy specification implies a desired property. For example, consider extending the amended list specification with the following definition of a prefix relation:

```
14  pred prefixes[ pre: List → List ] {
15     all l: List | Nil in l.pre
16     all l: List − Nil | l not in Nil.pre
17     all l, p: List − Nil | (p in l.pre) iff (l.car = p.car and p.cdr in l.cdr.pre)
18  }
```

The keyword **pred** introduces a parameterized fact or a *predicate*. The above predicate constrains its input relation to map every List to all Lists that are its prefixes. In particular, Nil is a prefix of every list (line 15); non-empty lists are not prefixes of Nil (line 16);

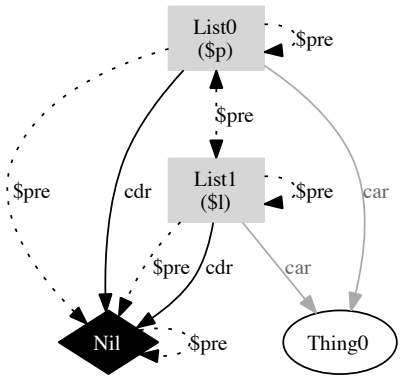
and, a non-empty list p is a prefix of l iff they have the same car, and the cdr of p is a prefix of the cdr of l (line 17).

A prefix relation on lists should be anti-symmetric—if two lists are prefixes of each other, then they must be equal. To check this, we execute the following command:

```
19  check {
20    all pre: List → List | all l, p: List |
21      (prefixes[pre] and p in l.pre and l in p.pre) implies p = l
22  } for 3
```

The check is performed by searching for a finite instance of the specification that satisfies the negation of the checked property. Such an instance, if found, is called a *counterexample*, and it shows a state that violates the property but is allowed by the specification.

The counterexample to our property is given bellow:



It demonstrates two distinct witness lists, $l and $p, that are nonetheless prefixes of each other according to a witness prefix relation $pre. The witness relations are so-called *skolem variables*, which are automatically created by the Analyzer for each existentially quantified variable. (The universally quantified variables pre, p and l in the **check** command become existentially quantified when the checked formula is negated.)

As the counterexample shows, the problem with the above formulation of anti-symmetry is the usage of *identity* instead of *structural equivalence* to compare the lists l and p. While $l and $p are not identical, they are, in fact, structurally equivalent. The Analyzer yields no counterexamples, even in a universe with up to eight lists and things, when checking the following, modified anti-symmetry property:

```
23  pred equivalence[ eq: List → List ] {
24    all a, b: List | a in b.eq iff (a.car = b.car and a.cdr in b.cdr.eq)
25  }
26  check {
27    all pre, eq: List → List | all l, p: List |
28      (prefixes[pre] and equivalence[eq] and p in l.pre and l in p.pre) implies p in l.eq
29  } for 8
```

Of course, a lack of a counterexample does not constitute a proof. The Alloy Analyzer can only perform finite simulations and checks, as its underlying constraint solving engine, called Kodkod, is based on a SAT solver. Exactly how the engine works, both as an

instance finder and a core extractor, is described elsewhere (Torlak and Jackson, 2007; Torlak et al., 2008; Torlak, 2009). We note here only that Kodkod takes as input a problem in *bounded relational logic*, which extends the logic of Alloy with a mechanism for specifying *partial instances*. A partial instance is simply the known part of a desired solution to a given set of constraints. For example, every Sudoku puzzle is a constraint solving problem for which the pre-filled cells form a partial solution. Kodkod is built to exploit partial instances for faster constraint solving, and it is thus applicable to large relational problems for which a solution is already partly determined—*e.g.*, course scheduling (Yeung, 2006) or network configuration (Narain et al., 2008). In contrast to Kodkod, the Alloy Analyzer takes as input the relational logic of Alloy, which has no notion of a partial instance. With the Analyzer, partial solutions must be encoded implicitly as additional constraints, resulting in a larger problem that is harder rather than easier to solve. Kodkod also differs from the Analyzer in that it is designed as a constraint solving API for use by automated tools, while the Analyzer is an IDE for interactive modeling and analysis of software systems.

3. Past and Present Applications

The Alloy toolset has been applied in a wide range of domains over the past decade. In this section, we focus on the three largest ones: modeling and analysis of software systems, bounded program verification, and test-case generation.

3.1. *Modeling and Analysis of Software Systems*

By far the most popular use for Alloy has been lightweight modeling and analysis of complex systems. Its expressive logic, automated analysis, and visualization capabilities provide a powerful combination of features for incremental building and exploration of system designs. The toy list example from Section 2 demonstrates, in miniature, the lightweight approach to modeling encouraged by Alloy. The user may start with a few key constraints; check that they capture relevant design properties in a desirable way; and then enrich the specification with additional details.

Some of the systems studied with Alloy include a library information system (Frappier et al., 2010); the flash filesystem (Kang and Jackson, 2008; Kang and Jackson, 2009); the Mondex electronic purse (Ramananandro, 2008); cryptographic protocols (Gassend et al., 2008); a proton therapy machine (Seater et al., 2007; Dennis et al., 2004); semantic web ontologies (Wang et al., 2006); and a multicast key management scheme (Taghdiri and Jackson, 2003). We discuss the three most recent studies below and refer the reader to the original papers for details on the rest.

The case study by Frappier *et al.* (2010) describes the modeling and analysis of a library information system using Alloy and five other tools—CADP (Garavel et al., 2007), FDR2 (Roscoe, 2005), NuSMV (Biere et al., 1999), ProB (Leuschel and Butler, 2003), and Spin (Holzmann, 2004). The system was specified in terms of ten possible events, and the specification checked against fifteen correctness requirements. Modeled events include the joining and leaving of members; the acquisition and discarding of books; and

lending, renewing, and returning of books by members. The specification was checked to ensure that it satisfies liveness properties, such as 'The library can always acquire a book that it does not already have', as well as safety properties, such as 'Only members are allowed to borrow books from the library.'

The study evaluated each of the modeling tools according to three criteria: ease of specifying the system; ease of specifying the properties to be checked; and the scalability of the analysis. The key limitation of Alloy was found to be its lack of built-in support for modeling of dynamic behaviors—expressing dynamic behaviors was possible, but not convenient. The B language (Abrial, 1996) supported by ProB turned out to be the best match for specifying information systems. Alloy's analysis, however, scaled significantly better than that of any other tool. The Analyzer was able to check all fifteen properties in a universe with 8 books and 8 members, in less than 5 minutes. For comparison, Spin scaled up to 5 books and 5 members, checking fourteen of the properties in over 2 hours. The remaining tools scaled up to 3 books and members, taking between a minute and an hour to check (most of) the properties.

Like Frappier *et al.*, Kang and Jackson (2008, 2009) found the support for dynamic modeling to be lacking in Alloy. Their case study focused on the modeling and analysis of a flash-based filesystem, including the modeling of flash hardware (Hynix Semiconductor et al., 2006) and of techniques (Gal and Toledo, 2005) for dealing with its limitations. The authors note that the absence of control constructs in Alloy made it cumbersome to specify multi-step operations, such as the write to flash memory.

The flash filesystem study made extensive use of Alloy's analysis capabilities. The correctness of the flash filesystem design was analyzed against the POSIX standard (The Open Group, 2003), by checking that the traces of a POSIX-compliant abstract filesystem subsumed the traces of the flash filesystem. While the analysis could not scale up to the size of a real file system, it nonetheless uncovered over 20 non-trivial bugs over the course of the design process. The final version of the design was checked in a universe with 24 data elements, which took approximately 8 hours to complete. The authors propose compiling the final design to code as a promising direction for future work.

Ramananandro (2008) also discovered subtle bugs with the help of Alloy, in a case study of the Mondex electronic purse system (Ives and Earl, 1997). The Mondex purse had been previously formalized in Z (Spivey, 1992) and proven correct by hand (Stepney et al., 2000). The Alloy specification of Mondex (Ramananandro, 2008) was derived from the Z specification, and checking it with the Analyzer against the hand-proven properties revealed three subtle bugs. Two of these were previously unknown errors in the proof of the properties, and one was a known bug in the Z specification itself. All three bugs were fixed in the Alloy formalization, but proving correctness of the amended specification had to be performed outside of the Alloy framework, using an external theorem prover.

3.2. *Bounded Verification*

Multiple efforts have been undertaken to apply Alloy directly to the analysis of code. These efforts share the same common approach: use relational logic to encode the claim that a procedure in a high-level programming language satisfies a specification, and use

Alloy (or its underlying engine) to search for counterexamples to that claim. A counterexample, if one exists, corresponds to a trace of the source code which violates the specification. Compared to testing, such an analysis could provide far greater code coverage and, therefore, a higher degree of confidence in the code's correctness. Unlike theorem proving, this approach is fully automated, but cannot provide a proof of correctness.

As with Alloy, to perform the analysis, the user must bound the problem. The bound consists of a limit on the size of each type in the procedure, and to ensure that traces are finite, the user must fix the maximum number of iterations around each loop and the maximum number of recursive invocations of a procedure. The soundness and completeness guarantees are the same as for Alloy: if a counterexample exists within the bound, one will be found; but if no counterexample exists within the bound, one may still lurk in a larger bound. For this reason, such style of code analysis has sometimes been termed *bounded verification* (Dennis, 2009).

The first bounded verification effort was Vaziri's Jalloy tool (Vaziri, 2004). Jalloy could check a method in Java against a specification of its behavior by translating the method to Alloy and invoking an early prototype of the Alloy Analyzer on the resulting constraints. In contrast to later approaches, Jalloy inlined the bodies of all called procedures directly into the procedure under analysis. Vaziri demonstrated the feasibility of this approach on individual methods of isolated data structures, but the approach could not yet scale to real programs consisting of multiple interacting components.

Building on Vaziri's work on Jalloy, Dennis built a successor tool called Forge (Dennis, 2009). At the time Forge was developed, the new Kodkod relational logic engine (Torlak, 2009) was already available, and Forge exploited its advantages over the previous Alloy Analyzer to great benefit. In particular, Forge employed a new translation from procedural code to relational logic involving symbolic execution, which was infeasible prior to the existence of the Kodkod APIs.

With Forge, Dennis introduced the Forge Intermediate Representation (FIR), a new interemediate representation language to support bounded verification. To analyze a method in a high-level programming language, Forge required it first be translated to FIR. The Forge tool was bundled with a translation for Java, and Toshiba research worked independently on a similar tool for C (Sakai and Imai, 2009). The introduction of FIR benefited Forge developers by providing a separation of concerns that helped avoid complexity in the tool's development, and it benefited Forge users by lowering the bar to build bounded verification tools for other high-level languages.

Forge also provides a coverage metric based on the Alloy/Kodkod unsatisfiable core technology. When Forge fails to find a trace of the procedure that violates the specification, it is because Kodkod determined there were no solutions to the provided relational formula within the provided bounds. When this happens, Forge can ask Kodkod to provide an unsatisfiable core of the formula—a subset of the top-level clauses that are themselves unsatisfiable—which Forge can in turn translate back into statements in the code that were not "covered" by the analysis. Such statements could effectively be removed from the procedure and the resulting procedure would still satisfy the specification. This could happen, for example, if the bound on the analysis is too small, or if the specification against which the procedure is being checked is under-constrained.

In parallel with Dennis's work on Forge, Dolby, Vaziri and Tip were exploring an alternative bounded verification approach based on Kodkod (Dolby et al., 2007). Their tool, called Miniatur, translates Java directly to Kodkod, using an algorithm that slices the input program with respect to the property being checked. Miniatur also employs efficient encodings for integer values and large sparse arrays, enabling the analysis of code that manipulates both (*e.g.*, hash maps). The integer encoding is designed to take advantage of Kodkod's support for bitvector arithmetic, which is handled by bit-blasting. Applying Miniatur to a variety of open-source programs revealed several violations of the Java equality contract. With its specialized encoding of integers, the tool was able to perform all checks using 16-bit integer arithmetic—a significant improvement over other Alloy-based tools (which generally cannot scale beyond 4-bit arithmetic), but still limited by the bit-blasting approach of the underlying constraint solver.

Related to JForge and Miniatur is Galeotti's approach to bounded verification, which is based on DynAlloy (Galeotti, 2010). DynAlloy (Frias et al., 2005) is an extension to the Alloy modeling language that includes actions from dynamic logic. Their tool translates Java code to DynAlloy, which is in turn translated to Alloy and analyzed with the Alloy Analyzer. Their results demonstrate significant performance gains over JForge from adding symmetry-breaking predicates on Java fields.

Taghdiri's specification inference (Taghdiri, 2007) technique complements bounded verification approaches. The technique, embodied in a tool called Karun, allows a method to be analyzed via a translation to relational logic, without requiring either specifications to be written for the called methods, or the full body of those methods to be inlined. Instead, Karun infers partial specifications of the called methods from their implementations automatically. Karun was built on top of Forge, but it could in principle be used in conjunction with any of the bounded verification techniques described above.

Karun begins by performing an abstract interpretation of each called method to obtain an initial, conservative abstraction of its behavior (Taghdiri et al., 2006). It then replaces each method call with its abstraction, obtaining an abstracted version of the original method. From there, it follows a standard CEGAR (Counter-Example Guided Abstraction Refinement) approach. It checks the method against its specification using Forge's bounded verification. If the abstracted method satisfies the specification, then the original method will necessarily satisfy it, so Karun terminates. If, on the other hand, bounded verification finds a counterexample, the counterexample witnesses a pre-/post-state pair for each method call. For each of these method calls, Karun invokes the bounded verification again to search for an execution of the called method that conforms to the pre-/post-state pair previously witnessed. If there exists such an execution, then the counterexample is valid. If no such execution exists, the counterexample is invalid and the abstraction is refined.

To refine the abstraction, Karun queries the underlying Kodkod model finder for an unsatisfiable core of the analysis, which includes formulas generated from the called method's implementation that prohibit the existence of the execution. The formulas in the core are then conjoined to the abstraction of the method, and the analysis of the method is performed anew. This process continues until no counterexamples are found (the method is correct) or a valid counterexample is found (the method is incorrect).

In the worst case, the method calls are eventually refined to the entire behavior of the called methods—the equivalent of inlining the method call. Taghdiri found Karun to scale better than inlining when the specification being checked was a partial property of the method's behavior, in which case only a limited specification for each called method needs to be inferred.

While most Alloy-based approaches to bounded verification have focused on sequential code, Torlak, Vaziri and Dolby (Torlak et al., 2010) present an approach for checking concurrent programs against memory models specified in relational logic. A memory model is a set of axioms that describe which writes to a shared memory location any read of that location may observe. Specifications of memory models are usually supplemented with small multi-threaded programs, called litmus tests, which illustrate behaviors that the model allows or prohibits. Because a litmus test is intended to elucidate the formal specification of a memory model, it is critical that each test correctly allows or prohibits its prescribed behavior. MemSAT is a fully automated tool, based on Kodkod, for checking litmus tests against memory model specifications. Given a specification of a memory model and a litmus program containing assertions about its expected behavior, MemSAT outputs a trace of the program in which both the assertions and the memory model are satisfied, if one can be found. Otherwise, the tool outputs a minimal unsatisfiable core of the memory model and program constraints. MemSAT was used to find discrepancies between several existing memory models and their published test cases, including the current Java Memory Model (Manson et al., 2005) and a revised version of it (Ševčík and Aspinall, 2008). It was the first tool to handle the full specification of the Java Memory Model, which eluded several prior analysis attempts due to the large state space induced by the model's committing semantics.

3.3. *Test Case Generation*

In addition to static code checking, a number of tools have used the Alloy framework to generate test cases for programs. A key strength of such approaches is their ability to produce structurally complex test data (such as balanced binary search trees). TestEra (Marinov and Khurshid, 2001), for example, employs Alloy in a specification-based, black-box framework for testing of Java programs. It uses a method's pre-condition to generate all non-isomorphic inputs up to a bounded size. The method is then executed on each test input, and the output is checked for correctness using the method's post-condition. TestEra expects pre- and post-conditions in the Alloy language and relies on the Alloy Analyzer's instance enumeration capability to generate test inputs. It incorporates symmetry breaking formulas into the Alloy specification to efficiently generate only non-isomorphic inputs.

Whispec (Shao et al., 2007) builds on TestEra, but focuses on maximizing code coverage. In other words, it is an approach for specification-based, white-box testing. Using Kodkod, Whispec solves a method's preconditions to generate an initial test input. It then executes the method on that input and builds the symbolic path condition of the executed trace. Negating some of the taken branch conditions yields a new path condition, which is then conjoined with the pre-condition, and solved again using Kodkod.

This process is repeated until all feasible branches of execution are covered, up to a given length.

Unlike TestEra, which expects a complete specification of program inputs and generates test cases using a single execution of the Analyzer, Kesit (Uzuncaova and Khurshid, 2008) performs incremental test generation using Alloy. Kesit focuses on generating tests for products in a software product line, where each product is defined as a unique combination of features. It specifies features as Alloy constraints and solves them incrementally. That is, each call to the Analyzer solves only a partial specification. Generated test cases are refined using Kodkod's support for partial instance definition.

The Alloy Analyzer has also been used for testing database management systems (DBMS). Since Alloy's logic is relational, it provides a natural fit for modeling relational databases. ADUSA (Khalek et al., 2008), for example, applies Alloy in the context of query-aware database generation. It takes as inputs a database schema and an SQL query, translates them to Alloy, and uses the Analyzer to generate all bounded, non-isomorphic test databases. Since the query information is taken into account, the generated databases will cover nontrivial, meaningful scenarios for query execution. The Analyzer also produces the expected result of executing the given query on each database. This information is then used as a test oracle by ADUSA.

Another approach, by de la Riva *et al.* (2010), uses Alloy to generate query-aware test databases with respect to an SQL test coverage criterion, called SQLFpc (Tuya et al., 2010). Given a query and a database schema, the technique transforms SQLFpc coverage rules to test requirements, and models them as an Alloy specification. Consequently, any instance found by the Analyzer represents a test database that satisfies both the schema and the coverage rules for the target query. Existing reports indicate that the Analyzer's limited support for arithmetic and string-based expressions hinders its applicability to database generation, since many queries contain aggregate functions and string operations.

To further automate DBMS testing, Abdul-Khalek and Khurshid (2010) present a technique for generating valid SQL queries with Alloy. Given a database schema and the SQL grammar, the technique automatically generates an Alloy specification that captures the syntax of SQL queries over that database. This guarantees that any queries generated from the specification (using the Analyzer) are syntactically correct. Additional constraints are included to prune out queries that are not semantically correct.

4. Extensions and Future Directions

As we have seen in the previous section, efforts to apply Alloy have brought up a number of challenges and directions for future development. The key challenges include a lack of built-in support for modeling dynamic systems (Frappier et al., 2010; Kang and Jackson, 2009); a lack of an automated way to turn Alloy specifications into executable prototypes (Kang and Jackson, 2009); a lack of an automated tool for verifying specifications (Ramananandro, 2008); and limited support for numerical constraints (de la Riva et al., 2010). We believe that addressing these challenges, while staying true to the lightweight nature of the language and its analysis, will both facilitate current applica-

tions of Alloy and inspire future ones. In the remainder of this section, we present recent efforts to overcome all four limitations, and step closer toward the future of Alloy as a comprehensive framework for lightweight design, analysis and construction of systems.

4.1. *Adding Imperative Constructs to Alloy*

The Alloy language contains no built-in support for specifying dynamic systems. As Alloy's logic is powerful enough to support many encodings of dynamic behavior, there is no need for the language to prescribe any one in particular. The two most popular idioms for specifying dynamic systems are based on events and traces. The event-based idiom prescribes the definition of atomic event signatures, each of which encapsulates its own semantics; the trace-based idiom calls for the addition of time-stamps to dynamic information and defines the semantics of the system using logical predicates that reason about these time-stamps. Specifications that are primarily concerned with the selection and ordering of events are naturally suited to the first idiom; those that are primarily concerned with the ways in which data change over time are suited to the second.

Both idioms produce similar results in terms of scalability, and both can be used to write concise specifications. However, neither idiom provides good support for the *composition* of dynamic behaviors, and, since both idioms represent *ad-hoc* solutions, specifications written by different authors may differ enough to reduce readability. The *ad-hoc* nature of these idioms also means that the Alloy Analyzer cannot take advantage of information about the dynamic elements of the specification to improve scalability or provide additional capabilities.

Imperative Alloy (Near and Jackson, 2010) is a syntactic extension to the Alloy language for specifying dynamic systems. The extension provides a set of operators taken from imperative programming for defining dynamic actions, and gives these operators the standard operational semantics. Mutable signature fields are annotated as such; updates to these fields are performed using the familiar **:=** operator, and actions may be sequenced using the standard **;**, providing the compositionality that is difficult to express in existing Alloy idioms. The extension also provides loops, pre- and post-conditions, and temporal quantifiers to bound the extent of action execution. Specifications in Imperative Alloy can be translated, through symbolic execution, to specifications in standard Alloy that utilize the trace-based idiom for their dynamic elements. Analysis of the resulting specifications scales similarly to the analysis of hand-written specifications.

DynAlloy (Frias et al., 2005) also extends Alloy with constructs for specifying dynamic systems. Like Imperative Alloy, DynAlloy allows the user to specify systems in terms of imperative commands or declarative specifications. It differs from Imperative Alloy, however, in that its semantics is based on dynamic logic, and the extension is not designed to produce human-readable translations in standard Alloy.

4.2. *From Specifications to Code: Compiling Imperative Alloy*

In addition to providing syntactic support for dynamic modeling, one of the goals of Imperative Alloy was to explore the possibility of flexible automated support for refining

specifications into executable programs. The use of concepts from programming languages makes it simple to translate nondeclarative Imperative Alloy specifications into imperative programs. The compiler (Near, 2010) from Imperative Alloy to Prolog automates the translation process and extends it to specifications that contain *some* declarative features, with the intention of allowing the user to refine the original specification just until the compiler produces a Prolog program with sufficient performance. This approach to synthesizing programs from relational specifications stands in contrast to prior work (Krishnamurthi et al., 2008; Dougherty, 2009), which translates a stylized subset of pure Alloy into a functional program backed by a persistent database—an approach that is complicated by the need to define imperative semantics for Alloy's declarative constructs.

The Imperative Alloy compiler takes a more direct route to code. The key to its success is the fact that Imperative Alloy makes nondeclarative control flow explicit, allowing the compiler to translate nondeclarative parts of the specification directly into efficient programs. Since Prolog supports the same declarative constructs—with the exception of universal quantification and negation—as Alloy, the compiler uses this support to execute the remaining declarative parts of the specification. In general, this strategy produces programs whose performance is directly related to the declarativeness of the source specification.

The combination of analysis using the Alloy Analyzer and execution using the compiler from Imperative Alloy to Prolog leads to a new style of programming. The programmer begins by writing a declarative specification and then refines the specification into a program by replacing declarative elements with nondeclarative ones. Each refinement step can be checked for correctness using the Analyzer, and at each step the specification can be compiled into Prolog and tested against the programmer's performance requirements. As soon as the compiled program runs fast enough, the programmer is finished. Experience has shown that even specifications with significant declarative elements can perform fairly well, meaning that this style of programming could save programmers time while producing more readable programs with fewer bugs.

4.3. *Verifying Properties of Alloy Specifications*

The Alloy Analyzer provides a fully automatic, easy-to-use engine for checking and simulating Alloy specifications. In the checking mode, the Analyzer looks for an instance that satisfies the specification, but violates a property of interest. Since the Analyzer translates Alloy constraints to propositional logic and solves them using a SAT solver, it can perform the analysis only with respect to a finite *scope*—an upper bound on the number of elements of each signature.

Consequently, although the Analyzer can produce counterexamples efficiently, it cannot automatically *prove* the correctness of a property. Under certain circumstances, a minimum scope can be computed so that correctness for that scope implies a general proof of correctness for any scope (Momtahan, 2005). Establishing the existence of a minimum scope, however, must be done outside of Alloy. As a result, proving the correctness of an Alloy specification requires an additional round of analysis in which the

user either rewrites the specification in the input language of a theorem prover for an interactive proof process (see, *e.g.*, (Ramananandro, 2008)), or establishes that the bound searched by the Analyzer was sufficient for a general proof. Such a two-phase analysis process requires significant effort from the user.

In order to overcome these limitations, some attempts have been made to verify Alloy specifications using interactive theorem provers. Dynamite (Frias et al., 2007) proves properties of Alloy specifications using the PVS theorem prover (Owre et al., 1992), via a translation to fork algebra. It introduces a PVS pretty-printer that shows proof steps in Alloy, reducing the burden of guiding the prover. Prioni (Arkoudas et al., 2003) integrates the Alloy Analyzer with the Athena theorem prover (Arkoudas, 2000). To overcome the challenge of finding proofs, Prioni provides a lemma library that captures commonly-used Alloy patterns.

More recently, SMT (SAT Modulo Theories) solvers have been used to prove properties of Alloy specifications (El-Ghazi and Taghdiri, 2011). SMT solvers are particularly attractive because they can efficiently prove a rich combination of background theories without sacrificing full automation. Compared to theorem provers that perform a complete analysis but require user interaction, SMT solvers are fully automatic, but may fail to prove quantified formulas. Recent SMT solvers, however, have shown significant advances in handling quantifiers (Ge and Moura, 2009; Bonacina et al., 2009; Ge et al., 2009). Furthermore, their ability to produce satisfying instances as well as unsatisfiable cores supports Alloy's lightweight and easy-to-use philosophy.

The SMT-based analysis mitigates the finite-analysis problem of the Alloy Analyzer by specifying all relational operators of Alloy as first-order SMT axioms; scope finitization is avoided altogether. However, since Alloy's logic is undecidable, the resulting SMT formulas can be undecidable, and thus the SMT solver may return an instance that is marked as "unknown." This indicates that the instance may be spurious, and must be double-checked. But if the SMT solver outputs "unsat," the input formula is guaranteed to be unsatisfiable, and the property it encodes is guaranteed to be correct. As a result, this approach complements the Alloy Analyzer: when the Analyzer fails to find a counterexample, the SMT-based analyzer can then translate the constraints to an SMT logic, aiming to prove the correctness of the property of interest. The user thus has the benefit of both sound counterexamples, provided by the Analyzer, and sound proofs, provided by SMT solvers.

5. Conclusion

We have presented an overview of Alloy, a language and a tool for the application of lightweight formal methods. The simplicity and expressiveness of Alloy's logic, and the automation of its analysis, have motivated its use in a variety of domains, ranging from classic formal modeling applications to bounded code verification, test-case generation, counterexample generation, and declarative configuration. The applications of Alloy have in turn influenced its development, inspiring a number of language extensions and alternative analyses. We have reviewed some of these applications, discussing the strengths and limitations of the Alloy approach. We have also presented recent efforts to address

the key challenges in using Alloy as a comprehensive framework for lightweight design, analysis and construction of systems. These efforts, we believe, are a roadmap to the future of Alloy: adding imperative constructs to the language while preserving its declarative nature; enabling verification of properties while keeping the analysis automatic; and adding tool support for turning Alloy specifications into implementation prototypes.

References

Abdul Khalek, S. and Khurshid, S. (2010). Automated SQL query generation for systematic testing of database engines. In *ASE '10*, pages 329–332.

Abrial, J.-R. (1996). *The B-Book: Assigning Programs to Meanings*. Cambridge University Press.

Arkoudas, K. (2000). *Denotational Proof Languages*. PhD thesis, Massachusetts Institute of Technology.

Arkoudas, K., Khurshid, S., Marinov, D., and Rinard, M. (2003). Integrating model checking and theorem proving for relational reasoning. *RELMICS*, pages 21–33.

Biere, A., Cimatti, A., Clarke, E. M., and Zhu, Y. (1999). Symbolic model checking without BDDs. In *TACAS'99*, pages 193–207.

Blanchette, J. and Nipkow, T. (2009). Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *TAP '09*.

Bonacina, M. P., Lynch, C., and Moura, L. (2009). On deciding satisfiability by DPLL and unsound theorem proving. In *CADE*, pages 35–50.

Chang, F. (2007). Alloy analyzer 4.0. http://alloy.mit.edu/alloy4/.

Chen, P. P.-S. (1976). The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36.

de la Riva, C., Suárez-Cabal, M. J., and Tuya, J. (2010). Constraint-based test database generation for SQL queries. In *AST '10*, pages 67–74.

Dennis, G. (2009). *A relational framework for bounded program verification*. PhD thesis, Massachusetts Institute of Technology.

Dennis, G., Chang, F., and Jackson, D. (2006). Modular verification of code. In *ISSTA '06*.

Dennis, G., Seater, R., Rayside, D., and Jackson, D. (2004). Automating commutativity analysis at the design level. In *ISSTA '04*, pages 165–174.

Dolby, J., Vaziri, M., and Tip, F. (2007). Finding bugs efficiently with a SAT solver. In *FSE '07*, pages 195–204.

Dougherty, D. J. (2009). An improved algorithm for generating database transactions from relational algebra specifications. In *RULE '09*, pages 77–89.

El-Ghazi, A. A. and Taghdiri, M. (2011). Relational reasoning via SMT solving. In *FM '11*.

Frappier, M., Fraikin, B., Chossart, R., Chane-Yack-Fa, R., and Ouenzar, M. (2010). Comparison of model checking tools for information systems. In *ICFEM '10*, pages 581–596.

Frias, M. F., Galeotti, J. P., López Pombo, C. G., and Aguirre, N. M. (2005). DynAlloy: upgrading Alloy with actions. In *ICSE '05*, pages 442–451.

Frias, M. F., Pombo, C. G. L., and Moscato, M. M. (2007). Alloy Analyzer+PVS in the analysis and verification of alloy specifications. In *TACAS '07*, pages 587–601.

Gal, E. and Toledo, S. (2005). Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37:138–163.

Galeotti, J., Rosner, N., Pombo, C., and Frias, M. (2010). Analysis of invariants for efficient bounded verification. In *ISSTA '10*.

Galeotti, J. P. (2010). *Software Verification using Alloy*. PhD thesis, University of Buenos Aires.

Garavel, H., Mateescu, R., Lang, F., and Serwe, W. (2007). CADP 2006: A toolbox for the construction and analysis of distributed processes. In *CAV*, pages 158–163.

Gassend, B., Dijk, M. V., Clarke, D., Torlak, E., Devadas, S., and Tuyls, P. (2008). Controlled physical random functions and applications. *ACM Trans. Inf. Syst. Secur.*, 10:3:1–3:22.

Ge, Y., Barrett, C., and Tinelli, C. (2009). Solving quantified verification conditions using satisfiability modulo theories. *AMAI*, 55(1):101–122.

Ge, Y. and Moura, L. (2009). Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *CAV*, pages 306–320.

Hammer, M. and McLeod, D. (1978). The semantic data model: a modelling mechanism for data base applications. In *SIGMOD '78*, pages 26–36.

Holzmann, G. J. (2004). *The Spin model checker*. Addison-Wesley.

Hynix Semiconductor, Intel Corporation, Micron Technology, Inc., Phison Electronics Corp., SanDisk Corporation, Sony Corporation, and Spansion (2006). Open NAND flash interface specification. Technical report, ONFi Workgroup.

Ives, B. and Earl, M. (1997). Mondex international: Reengineering money. Technical Report CRIM CS97/2, London Business School.

Jackson, D. (2006). *Software Abstractions: logic, language, and analysis*. MIT Press, Cambridge, MA.

Kang, E. and Jackson, D. (2008). Formal modeling and analysis of a flash filesystem in Alloy. In *ABZ'08*, London, UK.

Kang, E. and Jackson, D. (2009). Designing and analyzing a flash file system with Alloy. *Int J Software Informatics*, 3(2):129–148.

Khalek, S., Elkarablieh, B., Laleye, Y., and Khurshid, S. (2008). Query-aware test generation using a relational constraint solver. In *ASE '08*, pages 238–247.

Krishnamurthi, S., Fisler, K., Dougherty, D. J., and Yoo, D. (2008). Alchemy: transmuting base Alloy specifications into implementations. In *FSE '08*, pages 158–169.

Leuschel, M. and Butler, M. J. (2003). ProB: A model checker for B. In *FME*, pages 855–874.

Manson, J., Pugh, W., and Adve, S. V. (2005). The Java memory model. In *POPL '05*, pages 378–391.

Marinov, D. and Khurshid, S. (2001). Testera: A novel framework for automated testing of Java programs. In *ASE '01*.

Momtahan, L. (2005). Towards a small model theorem for data independent systems in alloy. *Electronic Notes in Theoretical Computer Science*, 128(6):37–52.

Narain, S., Levin, G., Kaul, V., and Malik, S. (2008). Declarative infrastructure configuration synthesis and debugging. *J. Netw. Syst. Manage., Special Issue on Security Configuration*.

Near, J. P. (2010). From relational specifications to logic programs. In *ICLP '10*.

Near, J. P. and Jackson, D. (2010). An imperative extension to Alloy. In *ABZ '10*.

Owre, S., Shankar, N., and Rushby, J. (1992). PVS: A prototype verification system. In *CADE-11*.

Ramananandro, T. (2008). Mondex, an electronic purse: specification and refinement checks with the Alloy model-finding method. *Formal Aspects of Computing*, 20(1):21–39.

Roscoe, B. (2005). *The Theory and Practice of Concurrency*. Prentice Hall, 3rd edition.

Sakai, M. and Imai, T. (2009). CForge: A bounded verifier for the c language. In *The 11th Programming and Programming Language workshop (PPL'09)*.

Seater, R., Jackson, D., and Gheyi, R. (2007). Requirement progression in problem frames: deriving specifications from requirements. *Requirements Engineering Journal (REJ'07)*.

Shao, D., Khurshid, S., and Perry, D. (2007). Whispec: White-box testing of libraries using declarative specifications. In *LCSD'07*, Montreal.

Spiridonov, A. and Khurshid, S. (2007). Pythia: Automatic generation of counterexamples for acl2 using alloy. In *International Workshop on the ACL2 Theorem Prover and its Applications*.

Spivey, J. M. (1992). *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall.

Stepney, S., Cooper, D., , and Woodcock, J. (2000). An electronic purse: Specification, refinement and proof. Technical report, Oxford University Computing Laboratory, Programming Research Group.

Taghdiri, M. (2007). *Automating Modular Program Verification by Refining Specifications*. PhD thesis, Massachusetts Institute of Technology.

Taghdiri, M. and Jackson, D. (2003). A lightweight formal analysis of a multicast key management scheme. In *FORTE '03*, volume 2767 of *LNCS*, pages 240–256. Springer Berlin / Heidelberg.

Taghdiri, M. and Jackson, D. (2007). Inferring specifications to detect errors in code. *Journal of Automated Software Engineering*, 14(1):87–121.

Taghdiri, M., Seater, R., and Jackson, D. (2006). Lightweight extraction of syntactic specifications. In *FSE'06*, pages 276–286.

The Open Group (2003). The POSIX 1003.1, 2003 edition specification. http://www.opengroup.org/certification/idx/posix.html.

Torlak, E. (2009). *A constraint solver for software engineering: finding models and cores of large relational specifications*. PhD thesis, MIT.

Torlak, E., Chang, F., and Jackson, D. (2008). Finding minimal unsatisfiable cores of declarative specifications. In *FM '08*.

Torlak, E. and Jackson, D. (2007). Kodkod: A relational model finder. In *TACAS '07*.

Torlak, E., Vaziri, M., and Dolby, J. (2010). Memsat: checking axiomatic specifications of memory models. In *PLDI '10*, pages 341–350. ACM.

Tuya, J., Suárez-Cabal, M. J., and de la Riva, C. (2010). Full predicate coverage for testing SQL database queries. In *Softw. Test. Verif. Rel.*

Uzuncaova, E., Garcia, D., Khurshid, S., and Batory, D. (2008). Testing software product lines using incremental test generation. In *ISSRE '08*.

Uzuncaova, E. and Khurshid, S. (2008). Constraint prioritization for efficient analysis of declarative models. In *FM '08*, Turku, Finland.

Vaziri, M. (2004). *Finding Bugs in Software with a Constraint Solver*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA.

Ševčík, J. and Aspinall, D. (2008). On validity of program transformations in the java memory model. In *ECOOP '08*, pages 27–51.

Wang, H. H., Dong, J. S., Sun, J., and 0001, J. S. (2006). Reasoning support for semantic web ontology family languages using Alloy. *Multiagent and Grid Systems*, 2(4):455–471.

Yeung, V. (2006). Declarative configuration applied to course scheduling. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA.