Alcoa: The Alloy Constraint Analyzer

Daniel Jackson, Ian Schechter and Ilya Shlyakhter
Laboratory for Computer Science
Massachusetts Institute of Technology
dnj@lcs.mit.edu

ABSTRACT

Alcoa is a tool for analyzing object models. It has a range of uses. At one end, it can act as a support tool for object model diagrams, checking for consistency of multiplicities and generating sample snapshots. At the other end, it embodies a lightweight formal method in which subtle properties of behaviour can be investigated.

Alcoa's input language, Alloy, is a new notation based on Z. Its development was motivated by the need for a notation that is more closely tailored to object models (in the style of UML), and more amenable to automatic analysis. Like Z, Alloy supports the description of systems whose state involves complex relational structure. State and behavioural properties are described declaratively, by conjoining constraints. This makes it possible to develop and analyze a model incrementally, with Alcoa investigating the consequences of whatever constraints are given.

Alcoa works by translating constraints to boolean formulas, and then applying state-of-the-art SAT solvers. It can analyze billions of states in seconds.

KEYWORDS

Object models, relational logic, constraint satisfaction, model checking, formal specifications, software analysis.

1 LANGUAGE FEATURES

Much of the novelty of Alcoa, in comparison to other analyzers, arises from two key features of its input language, Alloy [8].

First, Alloy is *relational*: its underlying data structures are sets and relations. This makes it easy to describe structures such as file systems, architectural topologies, naming schemes, etc. While relational structures can often be encoded with primitive datatypes, such as arrays and records, they can be expressed more succinctly, and analyzed more effectively, when relational notions are built-in. For example, in a file system description, the Alloy expression *Root.*children* & *Dir* might describe the set of directory objects reachable from the root directory. In languages without explicit support for relations, such a notion—if expressible at all—would require an explicit algorithm for computing the transitive closure.

Second, Alloy is *declarative*: the model is built by layering properties using conjunction, in contrast to operational languages in which the model is given by an abstract program. This allows partial models to be built, in which constraints describe how state components are related to one another, without explicit rules for how each component is updated. An Alloy model can thus be developed incrementally, using Alcoa at each step to investigate the consequences of constraints already given.

The reader might reasonable wonder why a new language was necessary. In short, by designing a language with analysis in mind, it was possible to offer a coherent collection of features that fit together in a simple manner. Defining a subset of a language such as Z [18], in contrast, would yield a language with ad hoc restrictions. Moreover, existing languages do not offer adequate hooks for tool support: Z, for example, does not distinguish syntactically amongst definitions, invariants, operations and assertions, which Alcoa treats differently from one another. The Object Constraint Language [20] of UML was considered as an input language, but its semantics are still unsettled, and it is not yet stable enough as a foundation for tool development.

2 ALCOA'S ANALYSIS

Alcoa provides two kinds of analysis, addressing the two principal risks of declarative modelling. The first risk is that the constraints given are too weak. Flaws of this sort are found by checking *assertions*, theorems that certain

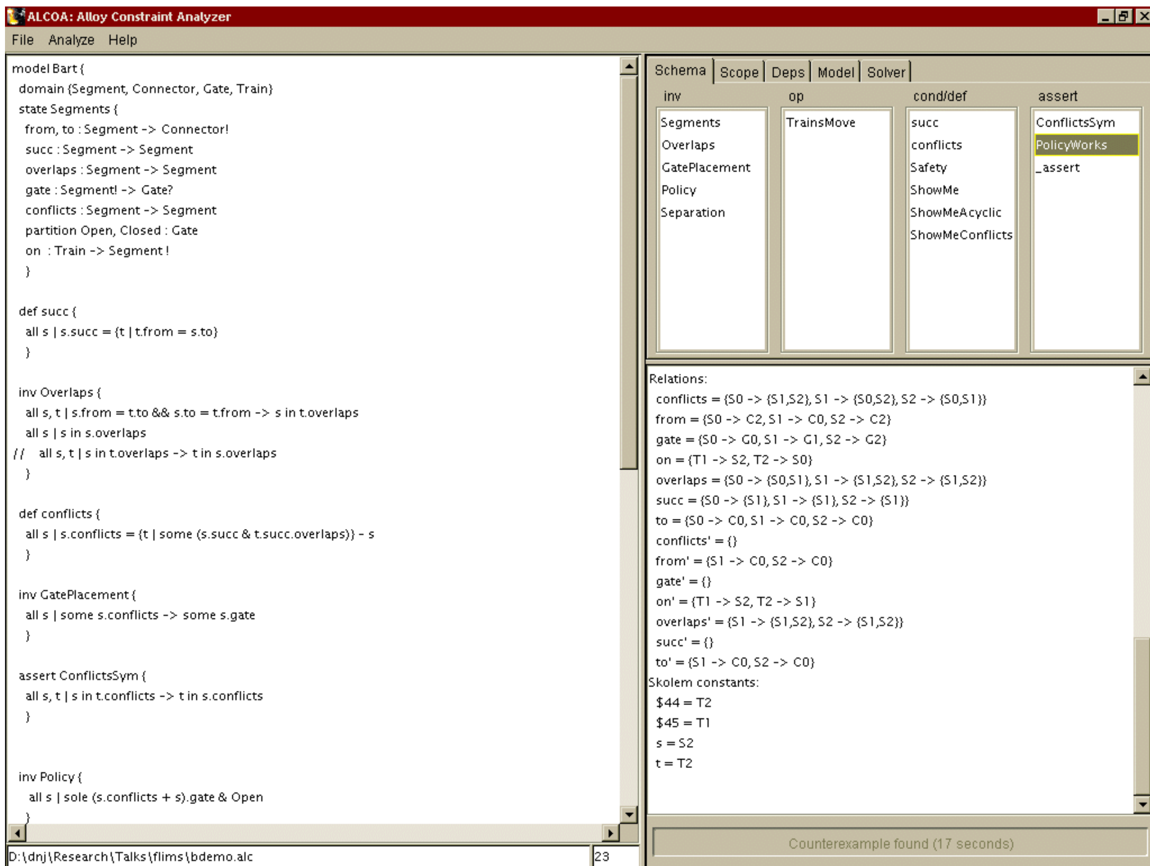

Figure 1: Alcoa screenshot

consequences follow from the constraints. Alcoa can check, for example, that an operation preserves an invariant, that one operation refines another, or that one invariant implies a second.

The second risk is that the constraints given are too strong; in the worst case, the constraints contradict one another and all states or transitions are ruled out. Flaws of this sort are found by exercising invariants or operations: ie, attempting to find satisfying states and transitions respectively.

Alloy is not a decidable language, so Alcoa cannot provide a sound and complete analysis. Instead, it conducts a search within a finite *scope* chosen by the user that bounds the number of elements in each primitive type. For example, an analysis of a file system might restricted to a scope that allows at most 6 file system objects. Alcoa's output is either an *instance*—a particular state or transition—or a message that no instance was found in the given scope. When checking an assertion, an instance is a counterexample and indicates that the theorem was not valid. When exercising an invariant or operation, an instance is a demonstration of consistency.

In theory, the user is not entitled to infer anything when no instance is found. In practice, however, if an instance exists, there is usually one in a small scope. So when no instance is found, there is a good chance that an assertion holds, or that an invariant is inconsistent.

Even in small scopes, the number of cases to consider is usually vast. A relation in a scope of k has $2^{k \times k}$ possible values; a model with only 3 relational state components in a scope of 3 thus has about a billion states. Of course many of these will be ruled out by constraints, and the search mechanism will prune away large parts of the space. In checking that an operation preserves an invariant, for example, the search might exclude most of the post-states that do not violate the invariant, thus considering only 'bad' executions of the operation, effectively executing it backwards. This scheme therefore can account for billions of possible executions of the operation, by ruling out in advance large classes that are not to cause problems.

3 HOW ALCOA WORKS

Alcoa is essentially a compiler. It translates the problem to be analyzed into a (usually huge) boolean formula. This formula is handed to a SAT solver, and the solution is translated back by Alcoa into the language of the model. The algorithm is described in [9]; an earlier version of the

translation scheme for an intermediate language without quantifiers appears in [10].

Alcoa comes with a suite of public domain SAT solvers whose parameters can be adjusted within Alcoa itself. Deterministic solvers based on the Davis-Putnam method [4], in particular SATO [21] and RelSAT [2], appear to work best.

4 MODE OF USE

The screenshot in Figure 1 shows Alcoa's user interface. On the left is an editing pane in which the model is created and modified. On the lower right, is a transcript pane to which Alcoa writes its output. The tabbed panes in the upper right allow the user to view the model in various intermediate forms, to change the scope and to select a SAT solver and set its parameters.

Alcoa is used as follows. First, a model is created and compiled. Compilation takes a couple of seconds, and finds superficl flaws, such as type errors. The user then selects a *schema*—a paragraph of the model to be analyzed—and starts a run. Alcoa responds either with an instance or a message that none was found. The user may then choose to edit the model, recompile and rerun, or to investigate the same schema further, by changing the scope or adjusting the solver parameters.

5 COMPARISONS

CASE tools focus on checking various forms of syntactic consistency (eg, use of names), and often include mechanisms for cooperative work and configuration management. Rudimentary code generation is often provided too, in which the model is construed not as an abstract description of the system but as an outline of its implementation. CASE tools developed for formal specification languages (such as Logica's *Formaliser* for Z [16]) usually provide type checking and pretty printing. None offer the kind of deep semantic analysis provided by Alcoa.

Model checkers have the same aim as Alcoa: to provide fully automatic semantic analysis. Model checkers are primarily designed, however, for addressing the complexities that arise from concurrency, and their input languages therefore offer parallel composition and communication mechanisms, and logics for describing event and state sequences. Alcoa, in contrast, addresses the complexity that arises from relational state structure. Moreover, the input languages of model checkers are usually abstract programming languages and are not designed for declarative specification; SPIN's language, Promela [6], for example, is a C-like language with guarded commands. Finally, model checking algorithms traverse the space of reachable states, and can, unlike Alcoa, investigate elaborate temporal properties. When an invariant is found to be violated by an operation, Alcoa's counterexample may be a transition from an unreachable state; a model checker in contrast will generate only feasible traces. The Alcoa user may have to strengthen the invariant to eliminate the bogus counterexample. This limitation has a flipside; model checkers are intrinsically non-modular, and cannot analyze partial descriptions in which only some operations are specified.

Animation and testing tools such as IFAD's VDM Toolbox [1] require part of the model to be in an executable sublanguage. Invariants may be given declaratively, but they cannot be analyzed in their own right. Instead, the tool executes operations from states constructed by the user, and can check, for example, that the states that follow satisfy invariants. Alcoa typically considers many more executions; while an animation tool is usually stepped through manually, Alcoa searches all possible executions— usually billions—within the given scope. Alcoa can be induced to behave like an animation tool. The user may specify a condition, for example, and then request an execution of an operation that starts from a state satisfying the condition; she may equally choose to constrain the post-state and 'execute' the operation backwards.

Theorem provers can establish with certainty that the model has particular properties. In contrast, Alcoa can be viewed as a theorem 'refuter'. Since theorem provers operate symbolically, there is no restriction to a finite scope, and properties even of infinite systems can be proven. Modern theorem provers, such as PVS [17], make extensive use of decision procedures, and can thus automate many low level proof steps. Several theorem provers have been embedded in tools for specific languages (eg, Z/EVES [3] for Z and LP [5] for Larch) to bridge the gap between proof obligations and assertions in the specification language. Despite these advances, theorem provers are still tools for experts only, since complex proofs will often fail because of flaws in the proof strategy, rather than in the assertion being checked. So although they can provide far greater assurance than tools like Alcoa, they are often too demanding for everyday use.

6 MATURITY

Alcoa is the successor to our Nitpick tool [11]. It overcomes two serious defiencies of Nitpick: a lack of scalability, and an input language that did not include quantifiers. Alcoa can handle a model with 20 relational state components, and can perform analyses in a scope of 6 or more. Its input language, Alloy [8], is close enough to UML to make a transcription of an object model diagram into Alloy a trivial task, and close enough to Z and VDM to make many existing specifications amenable to automatic analysis for the first time.

We have been using Alloy since September 1998. Its first

applications were to the design of a financial analysis tool [7] and to the post-facto specification of the key invariants of an air-traffic control system component [14].

Alcoa was made publicly available in September 1999. It has mostly been used to analyze existing models. We recast in Alloy our model of a mobile internet protocol we had previously analyzed with Nitpick [12], and were able to analyze it more effectively in Alcoa. A description of aggregation properties of COM [19] was translated with ease from Z into Alloy; Alcoa generates in seconds the flaws that were originally found more painstakingly by hand. We have also been using Alcoa in its own development, analyzing strategies for pruning its search. Recently, we have developed a method based on Alcoa for finding bugs in code [13].

Alcoa has been used in courses to teach formal methods at several universities (Kansas State University, Carnegie Mellon, Rochester Institite of Technology and the University of Hawaii). A variant of Alloy has been adopted in the latest version of the undergraduate software engineering text used at MIT [15].

The next release of Alcoa will offer a more efficient analysis that exploits symmetry, and that in particular seems to be able to exhaust a large space more rapidly when no instances are present. It will also support a richer language, with more powerful composition mechanisms, sequential composition of operations, and implicit frame conditions. The new tool will also be available as an API for incorporation into other tools. Finally, we plan to provide graphical display of instances.

7 AVAILABILITY

Alcoa is freely available for Windows, Linux and Solaris from *http://sdg.lcs.mit.edu/alcoa*. The web site provides a suite of annotated examples, a description of the Alloy language, a comparison to Z and UML, and an FAQ.

REFERENCES

[1] Sten Agerhold & Peter Gorm Larsen. The IFAD VDM Tools: *Lightweight Formal Methods*. FM-Trends 1998: 326-329.

[2] R.J. Bayardo Jr. & R. C. Schrag. Using CSP look-back techniques to solve real world SAT instances. *Proc. of the 14th National Conf. on Artificial Intelligence*, 203–208, 1997.

[3] Dan Craigen, Irwin Meisels, and Mark Saaltink. Analysing Z Specifications with Z/EVES. In *Industrial-Strength Formal Methods in Practice*, J.P. Bowen and M.G. Hinchey (Editors), September 1999.

[4] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, Vol. 7, pp. 202–215, 1960.

[5] Stephen J. Garland & John V. Guttag, *A Guide to LP: the Larch Prover*, MIT Laboratory for Computer Science, December 1991. Also available as Research Report 82, Compaq Systems Research Center, Palo Alto, CA.

[6] Gerard J. Holzmann. The Model Checker Spin. *IEEE Transactions on Software Engineering, Special issue on Formal Methods in Software Practice*, Volume 23, Number 5, May 1997, 279-295.

[7] Joseph B. Irineo. *An Object-Oriented, Maximum-Likelihood Parameter Estimation Program for GARCH(p,q)*. Masters Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

[8] Daniel Jackson. *Alloy: A Lightweight Object Modelling Notation*. Technical Report 797, MIT Laboratory for Computer Science, Cambridge, MA, February 2000.

[9] Daniel Jackson. *An Automatic Analysis for a First-Order Relational Logic*. Submitted for publication. Available at: http://sdg.lcs.mit.edu/~dnj/publications.

[10] Daniel Jackson. An Intermediate Design Language and its Analysis. *Proc. ACM Conference on Foundations of Software Engineering*, Florida, November 1998.

[11] Daniel Jackson and Craig A. Damon. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. *IEEE Transactions on Software Engineering*, Vol. 22, No. 7, July 1996, pp. 484–495.

[12] Daniel Jackson, Yuchung Ng and Jeannette Wing. A Nitpick Analysis of IPv6. To appear, *Formal Aspects of Computing*.

[13] Daniel Jackson & Mandana Vaziri. *Finding Bugs in Code using a Relational Constraint Solver*. Submitted for publication. Available at: http://sdg.lcs.mit.edu/~dnj/publications.

[14] Seung (Albert) Lee. *Object Modelling Applied to CTAS*. Masters thesis Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

[15] Barbara Liskov and John V. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. To appear.

[16] Logica UK Ltd. *Formaliser: A Specification Support Tool*. Information at http://www.16.com/offerings/Formaliser.html.

[17] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering,* 21(2):107-125, February 1995.

[18] J. Michael Spivey. *The Z Notation: A Reference Manual*. Second ed, Prentice Hall, 1992.

[19] K.J. Sullivan, J. Socha and M. Marchukov. Using Formal Methods to Reason about Architectural Standards. *Proceedings of the International Conference on Software Engineering* (ICSE'97), Boston, Massachusetts, May 1997.

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

[21] Hantao Zhang. SATO: An Efficient Propositional Prover. *Proc. of International Conference on Automated Deduction* (CADE-97).