Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 144–153
http://www.floc-conference.org/ICLP-home.html

FROM RELATIONAL SPECIFICATIONS TO LOGIC PROGRAMS

JOSEPH P. NEAR [1]

[1] Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology
32 Vassar St. Cambridge, MA, USA

Abstract. This paper presents a compiler from expressive, relational specifications to logic programs. Specifically, the compiler translates the Imperative Alloy specification language to Prolog. Imperative Alloy is a declarative, relational specification language based on first-order logic and extended with imperative constructs; Alloy specifications are traditionally not executable. In spite of this theoretical limitation, the compiler produces useful prototype implementations for many specifications.

1. Introduction

This paper presents a compiler from declarative, relational specifications to Prolog programs, eliminating the need for manual implementation. I express specifications in Imperative Alloy [28], a language based on the combination of first-order logic with transitive closure and the standard imperative programming constructs. My compiler transforms these specifications to Prolog for execution. Prolog represents an appropriate target language, since it supports nondeterminism and provides a database for storing global relations; the compiler uses these features to simulate Alloy's relational operators, quantifiers, and classical negation.

The existing Alloy Analyzer is designed for the *verification* and *animation* of specifications. My compiler is intended to complement the Analyzer by *executing* specifications. Animators perform their analyses within a fixed universe of predetermined size, while execution engines allow the creation of new objects. In practice, animators typically deal with models containing tens of objects, while execution engines must handle hundreds or thousands. In this case, this increased scalability comes at the cost of analysis: the Alloy Analyzer is designed to check all cases within a small bound, while my compiler executes a single, potentially large, case. In exchange, the compiler provides efficiency: most specifications can be executed fast enough to serve as prototype implementations.

Along with the Alloy Analyzer, my compiler provides end-to-end support for specifying and implementing programs. The Alloy language provides the expressive logic and relational constructs needed to express complex properties of programs and data; the Analyzer supports the animation and verification of program specifications; and the compiler presented in this paper allows for the efficient execution of those specifications.

Key words and phrases: logic programming, specification languages, executable specifications.

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.144

2. The Alloy Language

Alloy [16] is a modeling language based on first-order relational logic with transitive closure. It is designed to be simple but expressive, and to be amenable to automatic analysis. Imperative Alloy [28] adds imperative constructs, including assignment to global relations, sequential composition, and loops. The Alloy Analyzer is a tool for automatic analysis of Alloy models. While this analysis is bounded, it does allow for incremental, agile development of models; and the *small-scope hypothesis* [4]—which claims that most inconsistent models have counterexamples within small bounds—means that modelers may have high confidence in the results. The sacrifice of completeness in favor of automation is in line with the *lightweight formal methods* philosophy [17].

Alloy's universe is made up of uninterpreted atoms, each of which belongs to one of the disjoint sets defined using *signatures*. Signatures also may define global relations in the form of fields. As an example, consider a filesystem made up of file and directory nodes.

```
sig Data {}
abstract sig INode {}
sig DirNode extends INode { files: Name →INode }
sig FileNode extends INode { data: dynamic Data }
```

Directory and file nodes extend nodes, which are abstract, meaning that the file and directory nodes exhaustively partition the set of nodes. The " files " relation contains 3-tuples of type DirNode→Name→INode, while "data" is a mutable relation of type FileNode→Data. A representation of path names as linked lists of names can be defined similarly.

```
sig Name {}
abstract sig FilePath { name: Name }
sig DirName extends FilePath { dnext: FilePath }
sig FileName extends FilePath {}
```

Given a path name and a filesystem, a logical operation is to navigate through the filesystem to the node corresponding to the path name. I define a single step of this operation as an action in Imperative Alloy, using a singleton signature with mutable fields to hold pointers into the path name and the filesystem, as well as some temporary data.

```
one sig MVar {
    path: dynamic FilePath, current: dynamic INode, mdata: dynamic Data
}
action navigate {
    MVar.path := MVar.path.dnext;
    MVar.current := (MVar.path.name).(MVar.current.files) }
```

In defining navigation, I have used both imperative (field update and sequencing) and declarative (Alloy's generalized relational join) features of the language. Now, I can define reading from and writing to the filesystem by repeating "navigate" until the file node is reached and then either reading to or writing from the temporary storage.

```
action read {
    loop { navigate[] } && after MVar.current in FileNode;
    MVar.mdata := MVar.current.data
}
```

```
action write {
    loop { navigate[] } && after MVar.current in FileNode;
    let  file  = MVar.current | file.data := MVar.mdata
}
```

Given both actions, the user might wish to verify that writing to the filesystem and then reading from it produces the written data. I can define this property as an assertion to be checked by the Alloy Analyzer. In addition to the first-order quantifiers "**some**" and "**no**," I use the temporal quantifier "**always**" to indicate that the property must hold in all possible executions of the action. I use "**before**" and "**after**" to represent pre- and post-conditions on the action. The overall property states that for all starting nodes in the filesystem, it is always the case that if I begin at that node and write to the filesystem, remember the written data, reset the current node, and read from the filesystem, then the read data will match the remembered data.

```
one sig Temp { tdata: dynamic Data }
assert readMatchesPriorWrite {
  all n: INode |
    always |
      before (MVar.current = n && no f: FileNode | f.data = MVar.mdata) &&
      write; Temp.tdata := MVar.mdata, MVar.current := n;
      read ⇒ after (Temp.tdata = MVar.mdata) }
```

For more information on Alloy, refer to [16]; for more on Imperative Alloy, see [28].

3. Compiling Alloy Specifications

Any execution strategy for the Alloy language must allow relations as first-class values, nondeterminism, imperative constructs, and both relational and logical operators. I begin with Alloy's global relations, whose representation as dynamic predicates in Prolog I demonstrate using the filesystem from Section 2. For each signature, I generate a unary predicate representing membership in the signature and a predicate for each relation. I represent the existence of a single atom in each singleton signature by generating a fact.

```
:- dynamic sigName/1, sigFilePath/1, sigDirName/1, sigFileName/1,
           sigINode/1, sigDirNode/1, sigRootNode/1, sigFileNode/1,
           sigData/1, sigMVar/1, data/3, path/3, current/3,
           mdata/3, name/2, dnext/2, files/3.
sigRootNode(gensym62).
sigMVar(gensym63).
```

Relational values in Alloy may be thought of as sets of tuples. In Prolog, I can represent each tuple using a term; to represent the set of tuples in a relation, an expression may yield multiple instantiations of that term—one for each tuple in the Alloy relation. For example, I compile the expression representing the next element in a path as follows.

MVar.path.dnext → `sigMVar(MVar), path(T0, MVar, Path), dnext(Path, 0)`

The Prolog expression yields values by instantiating a member of the "MVar" signature, looking up a `Path` in the field of that "MVar," and then instantiating the free variable `0`

based on the next element of that path. The other relational operators can similarly be compiled into expressions involving Prolog's logical connectives.

I compile formulas involving relations to comparisons between their possible instantiations: $r_1 \subseteq r_2$, for example, assuming that r_1 and r_2 are binary relations, becomes `forall(r1(A,B), r2(A,B))`. Another option is to enumerate each relation's tuples explicitly (*e.g.* in a list or in the global database); this strategy may make lookup faster, but it forces the enumeration of the relational value of each subexpression, making the complex use of relational operators expensive.

Imperative Alloy also differs from Prolog in its imperative constructs: field update, sequential composition, and loops are notions built into the language. Sequencing and looping are easy to simulate in Prolog, and side effects can be expressed using `assert` and `retract`. Since Imperative Alloy's semantics call for side effects to interact well with nondeterminism, I have defined `assertl` to assert a list of terms, and then retract them upon backtracking, allowing side effects to be undone.

The combination of lazy evaluation and side effects means that a relation's value may depend on the value another relation had in the past. My prototype implementation therefore keeps track of the history of each global relation by adding an argument to each relation representing a *time-step*; the compiler passes the current time-step to called relations to instantiate the call's other arguments with the relation's value at that time-step. The "addr" relation, for example, has the type Time→Name→Addr, so I compile a reference to it to `addr(T, N, A)`, placing the time argument in the first position because most Prolog systems index on that argument. Parts of a relation's history upon which no "current" values depend may be eliminated in a process analogous to garbage collection.

This infrastructure makes compiling field assignments straightforward. I translate an update of the form $o.f := e$ by compiling e at the current time-step, then using `assertl` to update the global relation f. The first two arguments to f in the update are the next time-step and o; the remaining arguments are the free variables of the result of compiling e, and the body is the expression to which e compiles. I compile the full action for navigating the filesystem, for example, into a Prolog predicate as follows.

action navigate {
  MVar.path := MVar.path.dnext;
  MVar.current := (MVar.path.name).(MVar.current.files)
}
→

```
navigate(T0, T1) :-
  T2 is T0 + 1, sigMVar(Mv),
  assertl([((path(T2, Mv, Path) :- sigMVar(Mv2), path(T0, Mv2, Var3), dnext(Var3, Path))),
           ((data(T2, Var6, Var7) :- data(T0, Var6, Var7))),
           ((current(T2, Var8, Var9) :- current(T0, Var8, Var9))),
           ((mdata(T2, Var10, Var11) :- mdata(T0, Var10, Var11)))]),
  T1 is T2 + 1, sigMVar(Mv3),
  assertl([((current(T1, Mv3, Var22) :-
              sigMVar(Mv4), path(T2, Mv4, Var15), name(Var15, Var21),
              sigMVar(Mv5), current(T2, Mv5, Var20), files(Var20, Var21, Var22))),
           ((data(T1, Var24, Var25) :- data(T2, Var24, Var25))),
           ((path(T1, Var26, Var27) :- path(T2, Var26, Var27))),
           ((mdata(T1, Var28, Var29) :- mdata(T2, Var28, Var29)))]).
```

I compile each update in the navigation action to a call to `assertl` in Prolog and sequence them using conjunction. In both cases, the update itself is the first element in the list passed to `assertl`, and I form it by compiling the expression on the update action's right-hand side, then placing the result in the body of a rule defining the relation specified on the assignment's left-hand side. I also increment the current time-step so that the updated rule will be the sole definition of the relation at that time-step. The other elements passed to `assertl` represent the frame condition: in this new time-step, the other mutable relations do not change, so I generate rules to delegate these relations to their previous definitions.

I compile the action for reading from the filesystem in a similar way, except for its loop and declarative post-condition. I compile loops to nondeterministic repetition of an action, which I implement using the `loop` predicate. The post-condition checks that the current node is a file using the subset operator; in Prolog, this requires checking that the right-hand side of the "**in**" formula succeeds for every possible instantiation of the left-hand side.

action read {
  loop { navigate[] } && **after** MVar.current **in** FileNode;
  MVar.mdata := MVar.current.data
}
→

```
read(T3, T4) :-
  loop(T3, T5, navigate, []),
  forall((sigMVar(Mv), current(T5, Mv, Var33)), sigFileNode(Var33)),
  T4 is T5 + 1, sigMVar(Var39),
  assertl([((mdata(T4, Var39, Var38) :-
              sigMVar(Var35), current(T5, Var35, Var37), data(T5, Var37, Var38))),
           ((data(T4, Var40, Var41) :- data(T5, Var40, Var41))),
           ((path(T4, Var42, Var43) :- path(T5, Var42, Var43))),
           ((current(T4, Var44, Var45) :- current(T5, Var44, Var45)))]).
```

Finally, I compile the action for writing to the filesystem. Except for the "**let**" formula, it is nearly identical to that for reading.

action write {
  loop { navigate[] } && **after** MVar.current **in** FileNode;
  let file = MVar.current | file.data := MVar.mdata
}
→

```
write(T6, T7) :-
  loop(T6, T8, navigate, []),
  forall((sigMVar(Var47), current(T8, Var47, Var49)), sigFileNode(Var49)),
  sigMVar(Var51), current(T8, Var51, File), T7 is T8 + 1,
  assertl([((data(T7, File, Var55) :- sigMVar(Var54), mdata(T8, Var54, Var55))),
           ((path(T7, Var56, Var57) :- path(T8, Var56, Var57))),
           ((current(T7, Var58, Var59) :- current(T8, Var58, Var59))),
           ((mdata(T7, Var60, Var61) :- mdata(T8, Var60, Var61)))]).
```

This collection of predicates represents a simplified model of a filesystem that can be executed by a Prolog system; combined with a tool like FUSE (Filesystem in User Space), it can be used as a prototype implementation to store real data and be tested on an actual system. The user may add features slowly, using the Alloy Analyzer to verify their correctness.

C_E	::	expression $\rightarrow$ time $\rightarrow$ (Prolog expression, [variable])
$C_E(a,t)$ $(a \in vars)$	$\hat{=}$	$(\emptyset, [A])$
$C_E(f,t)$ $(f \in r)$	$\hat{=}$	$(f(A_1, A_2, ..., A_n), [A_1, A_2, ..., A_n])$ where f has arity n; $A_1, ..., A_n$ are fresh variables
$C_E(f,t)$ $(f \in r_d)$	$\hat{=}$	$(f(A_1, A_2, ..., A_n, t), [A_1, A_2, ..., A_n])$ where f has arity n; $A_1, ..., A_n$ are fresh variables
$C_E(e_1 \rightarrow e_2, t)$	$\hat{=}$	$((E_1, E_2), [A_1, ..., A_n, B_1, ..., B_n])$
$C_E(e_1.e_2, t)$	$\hat{=}$	$((A_n = B_1, E_1, E_2), [A_1, ..., A_{n-1}, B_2, ..., B_n])$ where $C_E(e_1, t) = (E_1, [A_1, ..., A_n])$ and $C_E(e_2, t) = (E_2, [B_1, ..., B_n])$
$C_E(e_1 + e_2, t)$	$\hat{=}$	$((A_1 = B_1, ..., A_n = B_n, E_1; A_1 = C_1, ..., A_n = C_n, E_2), [A_1, ..., A_n])$
$C_E(e_1 - e_2, t)$	$\hat{=}$	$((A_1 = B_1, ..., A_n = B_n, E_1, A_1 = C_1, ..., A_n = C_n, \backslash\texttt{+}E_2), [A_1, ..., A_n])$
$C_E(e_1 \& e_2, t)$	$\hat{=}$	$((A_1 = B_1, ..., A_n = B_n, E_1, A_1 = C_1, ..., A_n = C_n, E_2), [A_1, ..., A_n])$ where $C_E(e_1, t) = (E_1, [B_1, ..., B_n])$ and $C_E(e_2, t) = (E_2, [C_1, ..., C_n])$

Figure 1: Rules for Compiling Alloy Expressions into Prolog

4. Implementing the Compiler

My compiler transforms a complete Alloy specification into a Prolog program. In Alloy, sets are represented as unary relations; scalars, then, are singleton sets. For an Alloy expression whose value is an n-ary relation, my compiler produces a Prolog expression with n free variables; each possible instantiation of those free variables represents one tuple of the original relation. My compiler therefore produces a 2-tuple (e, v) containing the compiled Prolog expression e and a list of free variables v. I present the set of compilation rules for expressions in Figure 1; r represents the set of global relations in the original Alloy model, while r_d is the set of dynamic relations.

Two issues make compiling expressions tricky. First, the translation requires a representation of the time-step at which the expression is being evaluated. My implementation represents time-steps using integers; each global relation accepts one of these time-steps as its first argument and instantiates its other arguments to the values of the relation at that time-step. Second, some relational operators (*e.g.* difference) require the use of the cut or negation-as-failure. These impure elements restrict the contexts in which the compilation produces useful programs: the Alloy expression !(i $<$ j), for example, produces the Prolog expression `\+ (I<J)`, which will not correctly instantiate `I` or `J`.

Compiling Alloy formulas is straightforward, since Alloy's logical connectives map directly to those of Prolog. The equality and subset operators are the most interesting: since expressions evaluate to relations, both logical operators must examine *all* instantiations of the expressions' free variables generated by the resulting Prolog expressions. Figure 2 contains the rules for compiling formulas; again, the rules require the time at which the formula is being evaluated. Actions are compiled into formulas sequenced using conjunction. The

C_M	::	formula $\rightarrow$ time $\rightarrow$ Prolog expression
$C_M(e_1 \in e_2, t)$	$\hat{=}$	`forall(`$(E_1), (B_1 = C_1, ..., B_n = C_n, E_2)$`)`

$C_M(e_1 = e_2, t)$	$\hat{=}$	`forall(`$(E_1), (B_1 = C_1, ..., B_n = C_n, E_2)$`),`
		`forall(`$(E_2), (B_1 = C_1, ..., B_n = C_n, E_1)$`)`
		where $C_E(e_1, t) = (E_1, [B_1, ..., B_n])$
		and $C_E(e_2, t) = (E_2, [C_1, ..., C_n])$

$C_M(f_1 \&\& f_2, t)$	$\hat{=}$	$C_M(f_1, t)$ `,` $C_M(f_2, t)$
$C_M(f_1 \,\|\|\, f_2, t)$	$\hat{=}$	$C_M(f_1, t)$ `;` $C_M(f_2, t)$
$C_M(!f, t)$	$\hat{=}$	`\+` $C_M(f, t)$
$C_M(\textbf{all}\ x{:}\text{ite} \mid f, t)$	$\hat{=}$	`forall((`$x = A$`,`E`),` $C_M(f, t)$`)`
$C_M(\textbf{some}\ x{:}\text{ite} \mid f, t)$	$\hat{=}$	$x = A$`,`E`,`$C_M(f, t)$ where $C_E(e, t) = (E, [A])$

Figure 2: Rules for Compiling Alloy Formulas into Prolog

C_A	::	action $\rightarrow$ time $\rightarrow$ time $\rightarrow$ Prolog expression

$C_A(o.f{:=}e, t, t')$	$\hat{=}$	t' `is` $t + 1$`,`
		`assertl((`$f(o, A_1, ..., A_n, t')$ `:-` E`)),`
		`assertl((`$f(O, B_1, ..., B_n, t')$ `:-`
		`dif(`O, o`),` $f(O, B_1, ..., B_n, t)$`))),`
		$\bar{\forall} r : \textit{relations} \mid$ `assertl((`$r(O', C_1, ..., C_k, t')$ `:-`
		$r(O', C_1, ..., C_k, t)$`).`
		where $C_E(e, t) = (E, [A_1, ..., A_n])$

$C_A(a_1;\ a_2, t, t')$	$\hat{=}$	$C_A(a_1, t, T'')$`,` $C_A(a_2, T'', t')$ where T'' is a fresh variable
$C_A(a_1 \&\& a_2, t, t')$	$\hat{=}$	$C_A(a_1, t, t')$ `,` $C_A(a_2, t, t')$
$C_A(a_1 \,\|\|\, a_2, t, t')$	$\hat{=}$	$C_A(a_1, t, t')$ `;` $C_A(a_2, t, t')$
$C_A(action[e_1, ..., e_n], t, t')$	$\hat{=}$	`action(`$e_1, ..., a_n, t, t'$`)`

$C_A(\textbf{loop}\ \{act[e_1,...,e_n]\}, t, t')$	$\hat{=}$	E_1`,...,`E_n`, loop(`$t, t', act, [V_1, ..., V_n]$`)`
		where $C_E(e_i, t) = (E_i, [V_i])$ and
		`loop(T, T, F, Args).`
		`loop(T, Tp, F, Args) :-`
		`append(Args, [T, T1], A),`
		`apply(F, A), loop(T1, Tp, F, Args).`

Figure 3: Rules for Compiling Alloy Actions into Prolog

rule for field assignment updates the global relation f at the object o and time step $t + 1$ with the results of the right-hand side expression e. The next two lines of the rule express the frame condition: first, that the values of the relation f at objects other than o do not change, and second, that the values of the relations not being updated do not change. I use the "meta" quantifier $\bar{\forall}$ to represent quantification over the (static) set of relations in a given specification. Figure 3 contains the complete set of rules for compiling actions.

5. Related Work

As a notation, Imperative Alloy is similar to existing specification languages that support modeling dynamic systems [27, 23, 1, 5, 31, 20, 11, 9]; some of these notations also have associated analysis and animation tools. The novelty of this work is in the combination, using a single notation, of analysis and execution.

Most similar to my own work is an approach that translates Z specifications into Prolog [10], but produces relatively inefficient programs. Similarly, PVS has been translated to LISP [8] for execution, but the translation places restrictions on the PVS language. Squander [29] animates Alloy specifications embedded in Java programs, but provides the same level of performance as the Alloy Analyzer. My technique, on the other hand, produces programs that are fast enough to serve as prototype implementations. Notations for Abstract State Machines [22], rule-based transition systems [32, 30], concurrent object interactions [24], and the Maude language [7] have been translated to Prolog, but these are less expressive than Imperative Alloy. A Prolog translation also exists [25] from description logic, which has expressive power similar to that of Alloy. Many non-automated approaches [14, 21, 12] have been proposed, but the possibility of introducing errors during manual translation makes these unattractive, and most still require further refinement—even after a manual translation effort—for efficient execution.

Animation of expressive specifications is a well-studied topic. The toolkit supporting the B method [2] and tools for JML [6] and Z [19] all support animating specifications. However, many of these tools do not allow animation of the most expressive parts of the language, require a concrete instantiation of the initial state, and use constraint-solving approaches that do not scale as well as Prolog's search. By separating verification and animation from execution, my approach can provide both analysis and verification (using the Alloy Analyzer) and efficient execution (using my Prolog compiler).

The addition of constraints [18], more expressive logics [26, 15], more efficient execution strategies [3], and classical negation to logic programming [13] has made logic programming much more expressive; these advances may present an alternate approach to achieving the goal of a single language for specification and implementation.

6. Conclusions & Future Work

I have presented a compiler from the expressive, relational, first-order specification language Alloy to Prolog, making the process of implementing a specification automatic. Together with the Alloy Analyzer, this compiler represents a complete end-to-end solution for specifying and implementing programs. The Analyzer provides for the animation and verification of specifications, and the compiler I have presented in this paper allows for their execution.

My experience with this toolchain has identified two key areas for future work. First, the compiler does not yet identify parts of input specifications that may be troublesome to execute. Some Alloy constructs, such as negation and quantification, can make the resulting Prolog program impossible to execute. Specifications that make extensive use of these constructs do not seem to occur often in practice, but a warning message from the compiler would be useful to the user in case they do.

Second, performance of the compiled specifications is not yet optimal. With better knowledge of the strengths and weaknesses of the particular Prolog implementation the

compiler targets, I should be able to generate more appropriate code. Moreover, the compiler itself may be able to detect code that will perform poorly and signal a warning. I have already encountered cases requiring refinement of the specification in order to obtain efficient code; a profiling tool for a future version of the compiler might be able to suggest refinement in these cases.

Even without these improvements, my compiler produces useful prototype implementations for most specifications. My larger goal is to enable a single language to express systems at all levels of abstraction, from high-level requirements to low-level implementation. An implementation of such a language would naturally need to target several execution engines; more expressive features, such as first-order quantifiers, can be handled by the Alloy Analyzer, while low-level language features run at full speed. Of primary importance is the middle ground between these: the largest benefit to programmers comes from the use of expressive constructs in ways that can be identified and optimized by the compiler. By compiling Alloy to Prolog, I have shown that this middle ground is achievable: even the most expressive language constructs can be used in executable programs.

Acknowledgements

I am deeply grateful to Daniel Jackson, without whose guidance this work would not have been possible; to Eunsuk Kang, Rishabh Singh, and Jean Yang, who provided thoughtful comments on an early draft of this paper; and to the anonymous reviewers, who aided in the clarification of many points. This research was funded in part by the National Science Foundation under grants 0541183 (Deep and Scalable Analysis of Software), and 0707612 (CRI: CRD – Development of Alloy Tools, Technology and Materials), and by the Northrop Grumman Cybersecurity Research Consortium under the Secure and Dependable Systems by Design project.

References

[1] J.R. Abrial. *The B-book: assigning programs to meanings*. Cambridge Univ Pr, 1996.

[2] J.R. Abrial, M.K.O. Lee, D. Neilson, PN Scharbach, and I. Sørensen. The B-method. In *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development*, volume 2, pages 398–405. Springer, 1991.

[3] H. Ait-Kaci. *Warren's abstract machine: a tutorial reconstruction*. Citeseer, 1991.

[4] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the "small scope hypothesis". In *In Popl '02: Proceedings Of The 29th Acm Symposium On The Principles Of Programming Languages*, 2002.

[5] E. Börger and R.F. Stärk. *Abstract state machines: a method for high-level system design and analysis*. Springer Verlag, 2003.

[6] F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. Symbolic animation of JML specifications. *Lecture notes in computer science*, 3582:75, 2005.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and JF Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002.

[8] J. Crow, S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert. Evaluating, testing, and animating PVS specifications. *Computer Science Laboratory, SRI International, Menlo Park, CA, Tech. Rep., Mar*, 2001.

[9] G. Dennis, F.S.H. Chang, and D. Jackson. Modular verification of code with SAT. In *Proceedings of the 2006 international symposium on Software testing and analysis*, page 120. ACM, 2006.

[10] AJJ Dick, PJ Krause, and J. Cozens. Computer aided transformation of Z into Prolog. In *Z User Workshop: proceedings of the Fourth Annual Z User Meeting, Oxford, 15 December 1989*, page 71. Springer Verlag, 1990.
[11] MR Frias, JP Galeotti, CGL Pombo, and NM Aguirre. DynAlloy: upgrading alloy with actions. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 442–450, 2005.
[12] N.E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, 7(5):323–334, 1992.
[13] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Logic programming*, page 597. MIT Press, 1990.
[14] A. M. Gravell and P. Henderson. Why execute formal specifications? pages 165–184, 1991.
[15] J.S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.
[16] D. Jackson. *Software Abstractions: logic, language, and analysis*. The MIT Press, 2006.
[17] D. Jackson and J. Wing. Lightweight formal methods. *Lecture Notes in Computer Science*, 2021:1–1, 2001.
[18] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119. ACM New York, NY, USA, 1987.
[19] X. Jia. An approach to animating Z specifications. *COMPSAC-NEW YORK-*, pages 108–108, 1995.
[20] C.B. Jones. *Systematic software development using VDM*. Prentice Hall New York, 1990.
[21] A. Kans and C. Hayton. Using ABC to prototype VDM specifications. *ACM SigPLAN Notices*, 29(1):27–36, 1994.
[22] A. Kappel. Executable specifications based on dynamic algebras. In *Logic Programming and Automated Reasoning*, pages 229–240. Springer.
[23] Butler Lampson. 6.826 class notes, 2009. (`http://web.mit.edu/6.826/www/notes/`).
[24] P. Letelier, P. Sánchez, and I. Ramos. Prototyping a requirements specification through an automatically generated concurrent logic program. *Practical Aspects of Declarative Languages*, pages 31–45.
[25] G. Lukácsy and P. Szeredi. Efficient description logic reasoning in prolog: The dlog system. *Theory and Practice of Logic Programming*, 9(03):343–414, 2009.
[26] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51(1-2):125–157, 1991.
[27] C. Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(3):403–419, 1988.
[28] J. Near and D. Jackson. An Imperative Extension to Alloy. *Abstract State Machines, Alloy, B and Z*, pages 118–131, 2010.
[29] D. Rayside, A. Milicevic, K. Yessenov, G. Dennis, and D. Jackson. Agile specifications. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 999–1006. ACM, 2009.
[30] B. Schätz. Formalization and Rule-Based Transformation of EMF Ecore-Based Models. *Software Language Engineering*, pages 227–244, 2009.
[31] JM Spivey. *The Z notation: a reference manual*. 1992.
[32] Daniel Varro and Dniel Varr. Automated program generation for and by model transformation systems. In *Applied Graph Transformation (AGT'02), pages 161 - 174.*, 2002.