Towards Scaling Up DynAlloy Analysis using Predicate Abstraction

Rodrigo Ariño, Renzo Degiovanni, Raul Fervari,
Pablo Ponzio and Nazareno Aguirre

Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto,
Ruta 36 Km. 601, Río Cuarto (5800), Argentina.
{rodrigoarino, renzo.degiovanni}@gmail.com,
{rfervari, pponzio, naguirre}@dc.exa.unrc.edu.ar

Abstract. DynAlloy is an extension to the Alloy specification language suitable for modeling properties of executions of software systems. DynAlloy provides fully automated support for verifying properties of programs, in the style of the Alloy Analyzer, i.e., by exhaustively searching for counterexamples of properties in bounded scenarios (bounded domains and iterations of programs). But, as for other automated analysis techniques, the so called state explotion problem makes the analysis feasible only for small bounds. In this paper, we take advantage of an abstraction technique known as *predicate abstraction*, for scaling up the analysis of DynAlloy specifications. The implementation of predicate abstraction we present enables us to substantially increase the domain and iteration bounds in some case studies, and its use is fully automated. Our implementation is relatively efficient, exploiting the reuse of already calculated abstractions when these are available, and an "on the fly" check of traces when looking for counterexamples. We introduce the implementation of the technique, and some preliminary experimental results with case studies, to illustrate the benefits of the technique.

1 Introduction

With the well established success of model checking, computer scientists and an increasing number of practitioners have increased their interest in the construction of automated tools for the verification of software systems. Despite some breakthrough advances in automated analysis techniques, such as the use of symbolic representations in model checking, algorithms for automated verification essentially work by performing an exhaustive exploration of the state space of programs, and thus they can fail to terminate in a reasonable amount of time, even for simple models/programs and properties. Automated SAT based analyses, such as those associated with the Alloy Analyzer and bounded model checkers, are no exception to this situation, and therefore techniques for tackling the so called *state explosion problem* are necessary.
Alloy [13] is a specification language of increasing popularity in the last few years. Alloy is based on an extension of first order logic. Its main features are its

simplicity and ease of use, and the availability of an automated tool -the Alloy Analyzer- for simulating and finding violations of properties of Alloy specifications. As models of specifications (models in the sense of mathematical logic) are potentially infinite, the tool constructs all the possible instances up to a bound on the data domains given by the user, and checks if they satisfy the desired property. If an instance that violates the property is found, it is showed to the user as a counterexample. If, on the other hand, no counterexample is found, one cannot in principle guarantee the validity of the property being checked, since there might exist counterexamples to the property for bigger domains. However, in the absence of counterexamples, it definitely allows us to gain more confidence about the validity of the property.
While Alloy is useful for modeling static properties of systems, it has problems when dealing with the dynamics of software systems. To overcome these problems an extension to the language was proposed, called DynAlloy [8]. Based on dynamic logic, DynAlloy provides a simple way to specify properties of executions. DynAlloy also has an automated tool which allows users to find of specifications, by translating DynAlloy specifications into Alloy.
The usefulness of Alloy is justified by the small scope hypothesis [13], which asserts that most errors have counterexamples of small size. Some research supporting this hypothesis has been carried out [14]. Unfortunately, the analysis of Alloy and DynAlloy specifications do not scale up well, and analysis is only possible using small bounds. The reason for this problem is strongly related to the state explosion problem mentioned, and is due to the analyses being based on a reduction to boolean satisfiability, which is a well known NP problem.
In order to overcome the scalability issues associated with automated analysis techniques, many approaches have been proposed: abstraction [2, 5, 11, 6], symbolic execution [12, 7], static analysis, etc. We are interested in a predicate abstraction technique that allows us, given a set of predicates over the state space of a model, to automatically construct an abstract model whose states are boolean valuations of the predicates over concrete states. As the abstractions that we construct are conservative, the properties that can be verified on the abstract model also hold in the concrete model. Therefore, as the abstract model is usually simpler -in the sense that it has less states than the concrete model- less time and memory is required to complete the verification tasks.
A problem associated with abstraction is that we often construct abstract models that are too coarse to verify a certain property. For this reason, automatic algorithms for refining abstractions have been developed [6, 1]. They are based on the observation that when trying to verify a property over an abstract model two different kinds of counterexamples may appear: real counterexamples that when concretized perform a violation of the property on the concrete model, or spurious counterexamples, i.e., undesired behaviors introduced by the loss of information resulting from the abstraction. Given a spurious counterexample, the information it provides can be used to construct new abstraction predicates. The purpose of the new predicates is to refine the current abstract model, eliminating the spurious counterexample. The result of this may be that the new

refined model can be used to verify a property for which the original abstraction failed.
In [10] we introduced a fully automatic predicate abstraction algorithm for DynAlloy specifications, inspired by Graf and Saidi's work [11]. The algorithm traverses the abstract state space of the DynAlloy program in a depth first order. At each step an abstract state is constructed using only the previous abstract state and the specification of the action currently executed. A key aspect of our algorithm, is that previously constructed abstract states can be reused in the construction of new abstract states. As we use Alloy to automatically construct the abstractions, this has a big impact on the algorithm's run time performance. Now, we present a tool that fully implements that algorithm. This tool was developed with extensibility in mind, making it easy to add new abstraction algorithms, or modifications of the current one. This tool implements an adaptation of Das and Dill's technique [6] in order to deal with spurious counterexamples and refine abstractions.

2 The Alloy and DynAlloy Specification Languages

Alloy is a specificacion language based on relational logic, an extension of first order logic designed to support relational operators: relational image, closures, transpose, etc. Alloy syntax is OO-like and simple, it has only a few reserved keywords, making the language easy to learn and use for people with basic mathematical training. The language was designed with automatic tool support in mind, so it was developed together with an automatic tool -the Alloy Analyzer- which allows users to simulate models and search for counterexamples of properties that the model is intended to satisfy. Given an Alloy specification, a property, and bounds over the data domains of the Alloy model, the Analyzer constructs a propositional formula -representing the specification and the negation of the property- that is passed as input to an off-the-shelf SAT solver. If the SAT solver finds a model, a counterexample to the property is constructed based on this model. Otherwise the property does not have counterexamples on the given bounds.
Alloy is a model-oriented language, designed to specify properties of software systems. To model state change in Alloy it is necessary to introduce identifiers for states before and after the action is executed; this is a common practice in the Z language [15], on which Alloy was inspired. Although Alloy allows us to describe simple (single action) state change rather easily, it does not provide an adequate way to specify properties of more sophisticated executions of systems [8]. One way to simulate executions in Alloy is to manually introduce a notion of execution trace as a signature, which depends on the specification's signatures and operations and, therefore, must be defined on a per model (or per program) basis. This problem was addressed by an extension of Alloy -the DynAlloy language. DynAlloy introduces the notion of atomic action to model state change, and operations to compose these actions, whose semantics was inspired by dynamic logic. For the sake of space, we will introduce the most relevant parts of

the DynAlloy language by means of examples.
The state of a system is specified in Alloy and Dynalloy using signatures. Signatures define sets of elements, and relations between them. As an example, we define the structure of linked lists of integers:

```
  one sig Null {}       sig Node {                sig List {
                          next: Node+Null,          head: Node+Null,
                          value: Int              }
                        }
```

In the above definitions *Null* is a singleton set, representing the null reference, present in most programming languages. *Node* is the universe of nodes that may belong to a linked list, *next* is a binary relation that relates each node to its successor and *value* assigns to each node the integer value stored in it. Finally, *List* is the set of all possible lists, and *head* associates a head node (or a null reference) with each list.
DynAlloy atomic actions are useful for describing operations over the signatures of the model. They must be defined in terms of their corresponding pre and postconditions. For example, an operation that deletes the element at the head of a list can be defined as follows:

```
  action removeFirst[l: List] {
    pre { l.head != NullValue }
    post { l'.head = l.head.next }
  }
```

removeFirst can only be executed if the list's head is non null. Note the similarity of the dot expressions (like *l.head*) with that of object oriented languages. The postcondition introduces the variable *l'* to denote the state after the execution of the action. In this case, it states that the successor of the head of the list becomes the new head. In this way, we can think of the postcondition of an action as a relation between pre and post states.
DynAlloy provides three operators for composing atomic actions: sequential composition (;), non deterministic choice (+) and bounded iteration (*). Combining atomic actions and tests (assertions) using these operators we form DynAlloy programs. Programs are also annotated with pre and postconditions, to specify intended properties of programs, and then search for counterexamples. For example:

```
assertCorrectness removeAll[l:List]{
 pre = { }
 prg = {([l.head != NullValue]?;removeFirst(l))*;[l.head = NullValue]?}
 post = { l'.head = NullValue }
}
```

Assertion *removeAll* can be thought of as the specification corresponding to the program `while (head(l) != NullValue) do removeFirst(l)`. Its postcondition asserts that when the program finishes the list is empty, which should

obviously be valid.
Given a DynAlloy specification, the DynAlloy Translator automatically constructs an equivalent Alloy model which allows one to "execute" the program using the Alloy Analyzer. Bounds on the number of iterations to be executed and on the number of elements of signatures are necessary to perform the translation; the user must provide them. Executions of the program that violate the specification are shown to the user by the analyzer. It is worth noting that experiments showed that the analysis of dynamic properties is more efficient using DynAlloy (and its transation into Alloy) than the traditional Alloy approach, explicitly defining trace signatures in Alloy [9].

3 A Predicate Abstraction Algorithm for DynAlloy

Cousot introduced the idea of Abstract Interpretation [4], which consists of interpreting computations of programs over simpler (abstract) domains, that encode less information about the computations, but are usually easier to construct and explore. The method is based on the definition of two functions: $\alpha : \wp(S) \rightarrow S^A, \gamma : S^A \rightarrow \wp(S)$. α maps sets of concrete states to abstract states, and is called the abstraction function; γ associates with each abstract state the set of concrete states it represents. α and γ must conform a so called Galois Connection, that is, $\alpha(\gamma(s^A)) = s^A$ and $\varphi \Rightarrow \gamma(\alpha(\varphi))$; this ensures that abstract states represent over approximations of sets of concrete states. In addition, the sets of initial states must be included in the concretization of the initial abstract state, and applying an abstract transition τ_i^A to an abstract state s^A must result in a set of states containing $\tau_i(\gamma(s^A))$. This requirement ensures that each concrete trace has a a corresponding trace on the abstract model. The advantage of Abstract Interpretation is that the (safety) properties we verify on the abstract domain are also valid for the concrete domain. This technique is very powerful, allowing one, for instance, to apply model checking algorithms to infinite state systems, and to analyze systems with complex state spaces over which model checking algorithms would fail to terminate in a reasonable amount of time.

Two decades later, Graf and Saidi introduced Predicate Abstraction [11] as a way to automate the construction of abstract domains, given a set φ_1, ..., φ_l of predicates over the state of the program, that must be provided by the user. Based on this work we developed a predicate abstraction algorithm for DynAlloy [10]. Our concrete state space is composed of the DynAlloy signatures defining the state space of the program. Given abstraction predicates $\varphi_1(s)$, ..., $\varphi_l(s)$, our abstract state space consists of the set of monomials over boolean $B_1, ..., B_l$. A monomial is a conjunction of B_i's and $\neg B_i$'s, where each of the variables appear at most once. The boolean constant false is also considered as a monomial. Note that, to obtain the concrete set of states represented by an abstract state (monomial) a replacement of each B_i for the correspoding φ_i suffices. Hence, the concretization function is defined as:

$$\gamma(m) = m[B_1/\varphi_1, ..., B_l/\varphi_l]$$

To explain how to perform one step in the abstract execution of the program we first need to introduce the abstraction function:

$$\alpha(\psi(s)) = (\bigwedge\{B_i | all\ s : \psi(s) \Rightarrow \varphi_i(s)\}) \wedge (\bigwedge\{\neg B_i | all\ s : \psi(s) \Rightarrow \neg\varphi_i(s)\})$$

Thus, to automatically construct the abstract monomial corresponding to ψ we use the Alloy Analyzer to check whether ψ implies each of φ_i or $\neg\varphi_i$ for all concrete states s. Since we use the Alloy Analyzer for performing these checks, the answer in not necessarily absolute (we assume that a formula is valid if the Analyzer is unable to find bounded counterexamples for it). The bounds needed to run the checks must be provided by the user. To perform an abstract execution we first have to calculate the abstract state corresponding to the initial concrete states. We do this by directly applying α to the precondition of the DynAlloy program. Next, we have to calculate the abstract transitions and apply them in the order prescribed by the program. Thus, given an abstract state s^A and an abstract action τ_i^A, the idea to abstractly execute τ_i^A starting at s^A is to apply τ_i to each state of $\gamma(s^A)$, and then abstract away the result. Now, since DynAlloy postconditions are not predicates over the system state, but instead they relate action's initial states with final states, we need, in order to apply the abstraction function, to write a predicate for the strongest postcondition (SP) of τ_i with respect to $\gamma(s^A)$. That is, assuming τ_i has precondition $pre_{\tau_i}(s)$ and postcondition $post_{\tau_i}(s, s')$, $SP(\tau_i, \gamma(q))(s') = \forall s : pre_{\tau_i}(s) \wedge \gamma(q)(s) \wedge post_{\tau_i}(s, s')$. Now α can be applied to $SP(\tau_i, \gamma(q))(s')$ to obtain the desired abstract successor.

For example, if we consider the abstraction predicates φ_0 = NoLoops(l) and φ_1 = NullHead(l), an abstract execution of the RemoveFirst action (defined in section 2) starting at the monomial $B_0 \wedge \neg B_1$ will finish in the abstract state B_0.

It is interesting to note that our algorithm does not construct the complete abstract state space as the technique showed in[11] does, since this is -as some experiments have shown- very expensive. Instead, it constructs the abstractions "on demand", that is, the algorithm visits the program control tree in a depth first search manner, starting by abstracting the initial states, and then applying abstract transitions in the mentioned order. The DFS algorithm executes loops up to a bound given by the user. The postcondition of the DynAlloy program (concrete property to verify) is always included as an abstraction predicate (say φp). Therefore, as the procedure executes over approximations of concrete traces, we can ensure that if all the abstract traces end up in a monomial with B_p set then the concrete program satisfies the intended property (up to the bound for loops unrolling). If this is not the case, the algorithm will stop when it explores the first trace that does not satisfy the above requirement. These kind of traces are called abstract counterexamples, that is, they are traces that when concretized may or may not violate the concrete property. A concretized abstract counterexample that violates the concrete postcondition is called a real counterexample. Otherwise, it is a spurious counterexample: an abstract trace that does not have any concrete counterpart, produced by the loss of information on the abstraction process. Spurious counterexamples must be eliminated from the abstract model if we want to continue the abstract verification. Furthermore, they can be useful to refine abstractions, as it is exploited by the so called *counter example guided abstraction refinement* techniques.

An important feature of the algorithm is that (part of) the abstractions produced can be reused. As it builds the abstractions using the Alloy Analyzer, which is based on an NP-Complete algorithm, reusing as much information as possible is imperative to improve the running time of the algorithm. The idea behind reuse is that, if we have previously built an abstract state s'^A applying an action τ_i to another state s^A, then the boolean variables appearing at s'^A will be set in the same way on each state obtained by applying τ_i to any consistent abstract state s''^A that is stronger than s_A. This is due to the way we calculate the abstractions (by calculating logical implications). Hence, the algorithm stores the result of each execution of an abstract transition, and uses this information when possible at the following uses of the same action in the DFS execution.

Due to a lack of space, we will leave out the counterexample guided refinement process, and concentrate on the process for the construction of the abstraction (and checking its suitability).

4 Some Notes on the Implementation

In this section, we briefly describe the implementation of a tool automating the above described abstraction technique. The tool consists of two main modules, the abstraction module, that implements the abstraction algorithm, and the abstraction refinement module, that implements predicate discovery based on the spurious counterexamples returned by the abstraction phase.

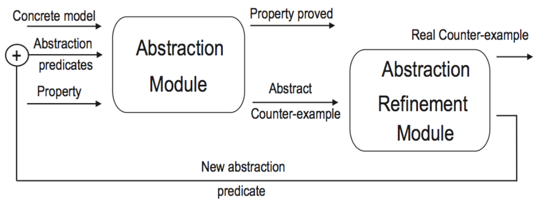

Fig. 1. Interaction of main tool's modules.

The interaction of these modules is depicted in figure 1. The abstraction module takes a concrete model and abstraction predicates as inputs, and it abstractly executes the given program. As a result, either the abstract model satisfies the specification, or an abstract counterexample trace is found. In the first case, the user is reported about the validity of the property and the whole execution ends. Otherwise, the counterexample found is passed as an input to the abstraction refinement module. If the counterexample is real, it is shown to the user and the process finishes; otherwise, the module uses the abstract trace to generate a new abstraction predicate. The new predicate is then added to the abstraction predicates and the abstraction process is restarted.

The tool is implemented using the Java programming language, and it interacts with the latest versions of the Alloy Analyzer and DynAlloy Translator, to perform certain tasks. As we mentioned, the tool was developed with extensibility in mind, so that new abstraction algorithms and new ways to perform predicate discovery can be incorporated straightforwardly. A diagram illustrating the design of the abstraction module is shown in figure 2. The main components of this module are the following:

– Main: retrieves the information from the GUI and it is responsible of executing the abstraction and the predicate discovery algorithms. Coordinates the interaction between the Abstraction Module and the Abstraction Refinement Module.
– IAbstractionModule: Interface that defines common operations to abstraction algorithms.
– AbstractionModule: Abstract class that contains attributes and implementations of common operations to abstraction algorithms for DynAlloy specifications. it contains attributes used to store input predicates, abstraction bounds, number of times loops should be executed, etc.
– OnDemandAbstraction: Implementation of the predicate abstraction algorithm discussed in section 3.
– IAbstractionUtils: Defines the signatures of the abstraction and concretization functions.
– AbstractStore: Stores the results of previously executed transitions, and provides methods to easily access this information, that will be used to avoid unnecessary calls to the Alloy Analyzer to improve the efficiency of the abstraction.
– AlloyUtils: Defines operations needed by the abstractor whose implementation is based on the Alloy Analyzer and the DynAlloy Translator source code.

5 Some Experimental Results

In this section, we present some experimental results using the presented tool. The experiments were carried out on an Intel Core 2 Duo of 2Ghz, with 2GB of RAM, running GNU/Linux 2.6. The version of the Alloy Analyzer employed was 4.1.8. We experimented mainly with two properties over the model of linked lists introduced before; these properties are: *(P1)* that the deletion method preserves the acyclicity of linked lists, and *(P2)* that no occurrences of the elements to be deleted belong to the list, before the current position of the cursor used for the implementation of deletion.

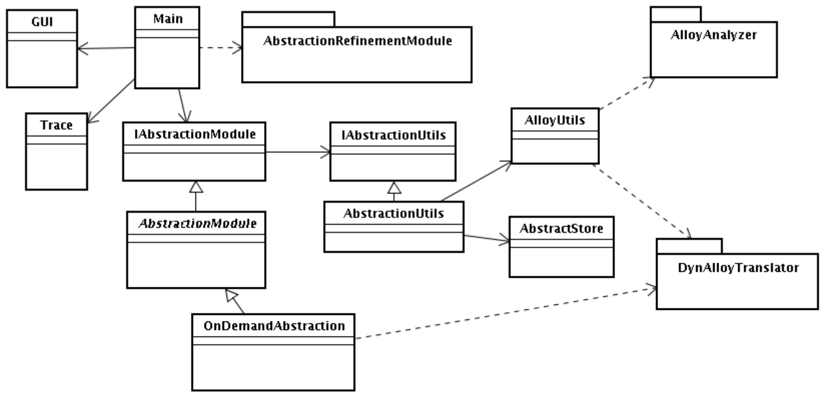

Fig. 2. Diagram for the abstraction module.

Without using abstraction, the Alloy Analyzer was able to verify P1 in 18 minutes and 36 seconds, for 20 loop unrolls and 21 as a scope for domains, and P2 in 27 minutes, for 15 loop unrolls and 16 as a scope for domains. The Alloy Analyzer exhausted the available memory for 24 loop unrolls and 25 as a scope for domains for P1, and 16 loop unrolls and 17 as domain scope for P2, and crashed after several hours running. Using abstraction via the presented tool, we were able to verify P1 in 4 minutes and 18 seconds for 20 loop unrolls and 21 as scope for domains (with 52428513 less calls to the Alloy Analyzer, thanks to reuse of abstraction calculations), and in 57 mins. and 26 seconds, for 25 loop unrolls and 26 as scope for domains (and a total of 1677721308 less calls to the Alloy Analyzer). For P2, we were able to verify the property for 20 loop unrolls and 21 as scope for domains in 4 mins. and 23 seconds, while it took the tool 76 mins and 18 seconds, for 25 loop unrolls and 26 as domain scope (the savings in calls to the Alloy Analyzer were similar to those of P1)

The results are promising, but we were unable to achieve the performance we expected. We are currently optimizing several parts of the abstraction tasks, aiming at scaling up DynAlloy analysis for about an order of magnitude.

6 Conclusions

We presented an implementation of a predicate abstraction technique for DynAlloy specifications, developed for scaling up the analysis of DynAlloy specifications. This tool enabled us to substantially increase the domain and iteration bounds in some case studies, and our implementation allows us to apply it automatically. For the sake of efficiency, our implementation exploits the reuse of already calculated abstractions when these are available, and an “on the fly” check of traces when looking for counterexamples. The preliminary results of some experiments we carried out are promising, showing that the technique is worthy. However, many improvements still need to be developed. First, there are several opportunities for saving space in the representation of the concrete and abstract computation trees, and we are currently moving to a control graph representation. Second, in its current form the tool can only handle DynAlloy specifications originating from code (where atomic actions are reversible); other more abstract DynAlloy models fail to be abstracted, due to technical reasons having to do with the calculation of weakest preconditions of traces when calculating or refining abstractions. We plan to enhance our tool so that these more abstract DynAlloy models are also handled.

References

1. T. Ball, B. Cook, S. Das and S. Rajamani, *Refining Approximations in Software Predicate Abstraction*, in Proc. of International Conference on Tools and Algorithms for the Construction and Analysis of Systems, LNCS, Springer, 2004.
2. E. Clarke, O. Grumberg and D. Long, *Model checking and abstraction*, ACM Transactions on Programming Languages and Systems, 16(5), ACM Press, 1994.
3. E. Clarke, D. Kroening, N. Sharygina and K. Yorav, *Predicate Abstraction of ANSI-C Programs using SAT*, Technical Report CMU-CS-03-186, Carnegie Mellon University, 2003.
4. P. Cousot, R. Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.*, Proc. of 6th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1977. ACM Press.
5. P. Cousot, *Abstract interpretation*, ACM Comp. Surveys, 28(2), ACM Press, 1996.
6. S. Das and D. Dill, *Counterexample Based Predicate Discovery in Predicate Abstraction*, in Proc. of International Conference on Formal Methods in Computer Aided Design, Portland, USA, LNCS, Springer, 2002.
7. Dennis, G., Chang, F, Jackson D. *Modular Verification of Code with SAT*. Proc. of the ACM/SIGSOFT Int. Symposium on Software Testing and Analysis, 2006.
8. M. Frias, J.P. Galeotti, C. López Pombo and N. Aguirre, *DynAlloy: upgrading alloy with actions*, in Proc. of the 27th International Conference on Software Engineering ICSE 2005, ACM Press, 2005.
9. M. Frias, J.P. Galeotti, C. López Pombo and N. Aguirre, *Efficient Analysis of DynAlloy Specifications*, in ACM Transactions on Software Engineering and Methodology (TOSEM), ACM Press, 2007.
10. N. Aguirre, M. Frias, P. Ponzio, B. Cardiff, J.P. Galeotti and G. Regis, *Towards Abstraction for DynAlloy Specifications*, in Proc. of the 10th International Conference on Formal Engineering Methods, LNCS, Springer, 2008.
11. S. Graf and H. Saïdi, *Construction of abstract state graphs with PVS*, in Proc. of 9th International Conference on Computer Aided Verification, Haifa, Israel, LNCS 1254, Springer, 1997.
12. S. Khurshid, C. Pasareanu, W. Visser, *Generalized Symbolic Execution for Model Checking and Testing*, in Proc. of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, LNCS, Springer, 2003.
13. D. Jackson, *Software Abstractions*, The MIT Press, 2006.
14. A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov., *Evaluating the "Small Scope Hypothesis"*, Technical Report MIT-LCS-TR-921, MIT CSAIL, 2003.
15. J. Woodcock, J. Davies, *Using Z: Spec., Refinement and Proof*, Prentice-Hall, 1996.