Combining B and Alloy

Leonid Mikhailov and Michael Butler
llm,mjb@ecs.soton.ac.uk

Declarative Systems and Software Engineering Group
Technical Report DSSE-TR-2001-2
February 2001

www.dsse.ecs.soton.ac.uk/techreports/

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton SO17 1BJ, United Kingdom

Combining B and Alloy

Leonid Mikhailov and Michael Butler
Department of Electronics and Computer Science
Univeristy of Southampton
Highfield, Southampton, SO17 1BJ
United Kingdom
llm,mjb@ecs.soton.ac.uk

February 2001

Abstract

In this paper we propose to combine two software verification approaches, theorem proving and model checking. We focus on the B-method and a theorem proving tool associated with it, and the Alloy specification notation and its model checker "Alloy Constraint Analyser". We consider how software development in B can be assisted using Alloy and how Alloy can be used for verifying refinement of abstract specifications. We demonstrate our approach with an example.

Keywords : B-method, Alloy.

1 Introduction

The approaches to creating verifiably correct systems can be divided in two broad categories: a top down approach when developers start with an abstract specification and gradually refine it to an executable implementation, which is guaranteed to be correct with respect to the specification, and a bottom up approach when developers attempt to implement a specification straight away and later on undertake a verification effort to make sure that their implementation complies to the specification.

The first approach is usually based on some sort of refinement calculus. Showing that a certain refined specification or, in fact, a final implementation complies to the corresponding abstract specification usually involves proving a lot of properties. Theorem proving is a very tedious process involving keeping in mind a multitude of assumptions and transformation rules. To help with this task a number of general purpose theorem provers exist, such as PVS, HOL, etc. [9, 4]. Such theorem provers usually have some automated tactics such as GRIND in PVS which attempt to prove the set goal automatically. As most of the refinement calculi (and or formalisations of programming notations) are formulated in undecidable logics (first and higher order logics) proving all goals is impossible. Thus the tool usually produces several subgoals that it didn't manage to resolve automatically and asks user guidance and assistance. The user by applying the set of rules and theorems available in the system attempts to prove the remaining goals.

With the second approach the specifiers usually formulate a number of liveness and safety properties that the implementation is supposed to comply to. It is, of course, possible to apply general purpose theorem provers for this purpose. However a different verification

technique, generally referred to as "model checking" is quite prominent with this approach. The general idea of model checking can be briefly expressed as follows: a program in its abstract representation, and the verification properties to be checked are formulated in some formalism based on logic. Next these formulas are submitted to the tool which tries to find a counter example violating the formulated verification conditions [7, 5, 8, 3].

Both theorem proving and model checking have advantages and disadvantages. The main advantage of theorem proving is that it permits to reason about infinite domains, and those are the most interesting in practice. A disadvantage is that a significant amount of highly qualified labour is required to verify even a relatively simple program. With theorem proving at times it can be difficult to say whether a property does not prove because it is simply not true or just some extra effort and ingenuity is required.

Model checking is much more applicable for finite domains, although there is a lot of ongoing research trying to apply this method to infinite domains. In general, for infinite domains, while model checking can find a counter example demonstrating that the specification is contradictory in one way or another, it cannot prove that the specification is correct. In this respect model checking is similar to testing, which also cannot prove the program correct. However what both of these approaches (model checking and testing) can do is to increase our confidence in the system. Another shortcoming of model checking is that it is usually applied for verifying consistency of a rather high level specifications, while ultimately everybody is interesting in the correctness of the software implementing these specifications. Obviously, while a specification can be perfectly correct, the implementation can be not. Verifying correctness of the executable programs with respect to their specifications is a topic of ongoing research.

In this paper we propose to combine these two approaches to verification, with the goal being to benefit from the advantages of both theorem proving and model checking. In particular we consider combining the B method and the corresponding tool with the Alloy specification notation and its constraint analyser. The B method is a top down development approach which is supported by industry-strength tools, which integrate a theorem prover for verifying the correctness of the specification and its refinements [1]. The Alloy specification notation is state-based and is supported by the Alloy constraint analyser, which is a finite state model checker [6, 7]. We briefly present these specification and verification methods in the following sections.

The main idea discussed in this paper is as follows. Complete formal proof of all proof obligations generated by the B tool is often practically infeasible. Often a proof obligations cannot be proved for the simple reason that it is not true. That can happen, for example, because a specification of an operation is not logically strong enough. Or, simply, the specification of an operation can be erroneous. The realization of impossibility to prove a certain proof obligations usually brings about a realization that certain amendments can be made to the specification, which would generate additional conjuncts in the hypothesis, permitting proof of the obligation. However, at times, proofs can be very tricky and although sufficient hypothesis are present, developer can experience difficulties proving them. Distinguishing between these two kinds of difficulties is important, as significant resources can be wasted on trying to prove goals that are simply not true.

Once the B tool has generated proof obligations we try to run an automated theorem prover supplied with the tool. It usually leaves some of the obligations unproved. Our idea is that before actually trying to prove this obligations interactively, we translate them into the Alloy language and run the Alloy constraint analyser on them. Counter examples that the Alloy constraint analyser can generate are usually suggestive, so that when a developer realizes how a certain instantiation of variables of the counter example invalidates the property under consideration, it becomes clear which amendments can be made to the specification to exclude the counter example. This suggest a certain debugging process, which most certainly has a

shorter cycle than when interactive prover is used for finding error. Once the Alloy constraint analyser cannot find a counter example for a sufficiently large instantiations of the domains, it is a good indication that the verified property is probably correct. The developer can then return to B interactive prover with confidence that this property should be possible to prove.

Translation between B specification notation and Alloy is manual at the moment. However, in case certain modification and additions would be made to the Alloy specification notation, such translation could be done automatically in both directions. We discuss such modifications and additions in this paper. To illustrate our approach we consider an example.

2 Summary of the Used Formalisms

Let us now briefly present the formalisms of B and Alloy and the development methods associated with them.

2.1 The B Specification and Verification Method

The B method has an associated specification notation, the so-called Abstract Machine Notation (AMN). This specification notation is classified as a state-based notation and is quite similar to such well-known formal notations as Z and VDM [10, 12]. The similarities between Z and B arise from the fact that the creator of the B method Jean-Raymond Abrial is also the author of Z. Compared to the specification notation of Z, AMN is more appealing to programmers, as it includes such statements as "**IF THEN ELSE**" and "**WHILE**" along with nondeteministic specification statements such as nondeterministic choice "**ANY**". The B method has three development stages: the specification, the refinement, and the implementation. Not all of the constructs of AMN are available at all the stages of the development cycle. We briefly introduce the necessary subset of AMN as we present the example.

The B method is supported by two commercially available tools, "B-Toolkit" developed and distributed by B-Core company, UK [2], and "AtelierB" developed and distributed by Steria, France [11]. In general, the tools are quite similar and each of them excels in slightly different aspects of the method. Accordingly, in the following discussion we refer to both of them as "the tool".

Development in the B method is centred around the concept of *machines*: an abstract machine – **MACHINE**, a refinement machine – **REFINEMENT** , and an implementation machine – **IMPLEMENTATION**. Machines are similar to modules encapsulating their internal representation and providing operations for manipulating this internal representation. The state of a machine can be accessed and modified by applying the operations defined in this machine.

The developer starts off with translating an informal specification into an abstract **MACHINE**, which is allowed to use only an abstract subset of all available statements in AMN. As a part of the abstract machine specification, the developer has to introduce an invariant, which should be established by initialization and should hold before and after the execution of all operations of this machine. When the developer submits the produced specification to the tool, it generates a number of theoretically justified verification conditions which are sufficient to establish that the specification is not contradictory, or consistent.

Next, the developer defines a **REFINEMENT** machine which, in general, is similar to the abstract specification machine, but is usually more deterministic, yet not directly translatable into a programming language like C or Ada. The refinement machine must include an invariant which usually consists of two parts, the part restricting the variables introduced in a refinement

step, and the so-called "gluing invariant" relating these variables and their counterparts in the corresponding abstract machine. When the refinement is submitted to the tool, the latter generates a number of proof obligations sufficient to establish that the refinement is consistent and that it correctly implements the corresponding specification. In general, a refinement machine does not have to refine an abstract machine, it can refine another refinement. In fact, usually the development process includes several refinements until an implementation machine is reached.

Finally, the developer must define an **IMPLEMENTATION** machine which maps directly to a programming language such as C or Ada. An implementation usually has a set of its own variables of certain pre-defined types supplied in the libraries with the tool. An implementation also must have its own invariant relating its variables with the variables of the last preceding refinement. Operations of the implementation must be expressed only using these implementation variables and relying only on the deterministic subset of AMN. Similarly to the case of refinement machines, the tool generates proof obligations for showing that the implementation is consistent and that it correctly implements the previous refinement. In this paper, we only focus on the features of abstract machines and refinements relevant to our discussion.

As soon as some proof obligations are generated, the developer can try to discharge them using an automated theorem prover incorporated in the tool, which attempts to discharge the generated proof obligations. Typically, there is a number of proof obligations that the automated prover cannot discharge, so the developer can switch the prover to the manual mode and attempt to prove the remaining proof obligations interactively.

2.2 The Alloy Specification and Verification Method

The Alloy specification notation and the Alloy Constraint Analyser are the research products of Daniel Jackson and his colleagues at MIT [6, 7]. The Alloy specification language (to which we further refer as Alloy) is also state-based like B. An Alloy specification usually contains several sections. One of the obligatory sections is for variable declaration, where variables can be declared as either atoms, subsets of declared domains, or relations of various kinds connecting these sets and/or domains. Declaration of the variables can be arranged so that the specification would have an implicit invariant restricting the set of possible states in which these variables can be present. In addition, in another section of the specification, the developer can write down an arbitrary number of named explicit invariants that further restrict the state. The developer can also write down a named assertion containing an arbitrary logical formula expressed on the variables of this specification. In yet another section of the specification, the developer can write down named operations modifying variables declared in the specification. Operation specifications describe a relation between pre- and post-states of the variables, similar to operation schemas of Z.

Verification with Alloy typically proceeds in the following manner. After the developer has recorded the variables and all implicit and explicit invariants restricting the set of states the variables can be in, he or she can write down some conjectures about the relation between the declared variables in the form of named Assertions. It is then possible to submit such an assertion to the Alloy constraint analyser which tries to find a counter example invalidating the assertion. The Alloy constraint analyser does this by converting the assertion, all related variable declarations, and appropriate invariants to a boolean formula, negating it and submitting it to one of several available general purpose boolean solvers. The chosen solver, in turn, tries to find an instantiation of the variables in the submitted formula making it true. Naturally, to make this process finite, the user of the Alloy constraint analyser is asked to indicate the dimensions of the participating domains.

MACHINE *DbAbstr*
SETS
STUDENTS **;** *GRADES*
VARIABLES
abstDb
INVARIANT
$abstDb \in STUDENTS \nrightarrow GRADES$
INITIALISATION
$abstDb := \{\}$
OPERATIONS
$append(\ st\ ,\ gr\ ) \ \widehat{=}$
PRE
$st \in STUDENTS \wedge gr \in GRADES \wedge st \notin \mathsf{dom}\ (\ abstDb\ )$
THEN
$abstDb := abstDb \cup \{\ st \mapsto gr\ \}$
END
END

Figure 1: The abstract machine *DbAbstr*

The developer can also verify the operations defined in the specification against any or all of the invariants. For this, the developer has to mark an operation he or she wants to verify against a particular invariant, and the analyser then tries to find an example instantiation of the variables which satisfies the invariant before an execution of the operation but does not satisfy it after. Internally, the analyser achieves this in a manner similar to verifying assertions.

We briefly introduce the subset of the Alloy specification language necessary for our purposes as we present the example.

It is important to mention that at the moment Alloy does not provide any support for verifying implementations or refined specifications on compliance with the original specification. In this paper we discuss how such features can be introduced to Alloy.

3 Example of Specifications in B and Alloy

In this section we follow the outline of our verification method briefly described in the introduction. Rather than discussing the method on an abstract level, we chose to demonstrate it with an example. Due to numerous restrictions and shortcomings of the Alloy specification notation, we chose a rather simple example of specifying a database of student grades. Yet, verifying this specification arises a multitude of interesting issues that we discuss below.

3.1 Specifying a Student Grades Database in B

Suppose that we would like to create a simple database containing information about students and their grades. On an abstract level, such a database can be modelled as a partial function. The B specification of such a model can be represented as an abstract machine *DbAbstr*, as shown in Fig.1.

This machine introduces two new domains, which are declared in the section **SETS**: *STUDENTS* and *GRADES*. These domains are the fixed sets sometimes referred to as *deferred sets*, as the developer only needs to give them a concrete representation in the implementation.

The next section of the B specification contains declarations of the variables, which hold the state of the machine. In our case, this is the variable *abstDb*.

The **INVARIANT** section holds the invariant of the machine. In general, an invariant is a predicate which is established by the initialization of state variables and holds before and after execution of all operations declared in the machine. In B, an invariant usually includes predicates that give a type to the state variables declared in the **VARIABLES** section. In our machine, *abstDb* is constrained to be a partial function from the deferred set *STUDENTS* to the deferred set *GRADES*.

In the next section **INITIALISATION**, all variables of the machine must be initialized. Thus, *abstDb* is assigned an empty set.

As follows from the name of the next section, it contains the definitions of all operations defined for this machine. To illustrate our idea, it is sufficient to provide only one operation. Therefore, the machine *DbAbstr* only has an operation **append**, for adding records about students' grades into the database. This operation has a precondition verifying the types of the corresponding parameters and also checking that the submitted student is not already in the database, i.e. in the domain of the partial function *abstDb*. In B, the outcome of an operation is only defined in those states where its precondition evaluates to true.

As soon as the definition of the *DbAbstr* machine is complete, we can run the type checker, the proof obligation generator, and the automated theorem prover on it. Because of the simplicity of *DbAbstr*, the automated theorem prover of the tool can resolve one hundred percent of the generated proof obligations.

Now let us consider a refinement of our student database. In this refinement, shown in Fig.2-3 we implement the student database as a connected list of nodes. The clause **REFINEMENT** declares that the machine is intended to be a refinement of another machine. In the next section of the refinement machine, the developer has to indicate which exactly machine it refines, in our case it is *DbAbstr*. Similarly to abstract machines, refinements can also declare deferred sets. In our case, we declare a new set *LINKS* that will serve as a domain of all links available for building a linked list. Next, the developers can declare some constants original to the refined specification, so we declare a constant *nil* that is used for marking the end of the list. The clause **PROPERTIES** is used for constraining the declared constants, in particular, the developers must indicate the type of the constants: *nil* is an element of the domain *LINKS*.

Next, we declare variables *stDb* ,*grDb* , *next* , and *head* that are used for implementing a linked list. As can be seen from the upper part of the invariant, *stDb* is declared as a partial injective function associating *LINKS* with *STUDENTS*. Note that, as the function is injective, there can be no two different links referring to the same student. On the other hand, *grDb* is declared not as injective function, but simply as a partial function from *LINKS* to *GRADES* – clearly, several students could have received the same grade on an exam. The function *next* represents the linked list itself, and is injective, which helps us later to state that the list is really linked, i.e. all of its nodes can be reached from its *head*.

The invariant in a refinement can, in general, be divided into three parts. The first one describes the types of the variables declared in the refinement. The second one describes the relations between the variables declared in the refinement that are true after the initialization of these variables and remain true before and after execution of all operations of this machine. In our case, this part of the invariant can be subdivided into three conjuncts. The first one states that the domains of *stDb*, *grDb* , and *next* are equal. This condition guarantees that

REFINEMENT *DbConcr*
REFINES *DbAbstr*
SETS
$LINKS$
CONSTANTS
nil
PROPERTIES
$nil \in LINKS$
VARIABLES
$stDb \ , \ grDb \ , \ next \ , \ head$
INVARIANT
$stDb \in LINKS \rightarrowtail\!\!\!\!\!\!\!+\; STUDENTS \ \wedge$
$grDb \in LINKS \;+\!\!\!\!\!\rightarrow GRADES \ \wedge$
$next \in LINKS \rightarrowtail\!\!\!\!\!\!\!+\; LINKS \ \wedge$
$head \in LINKS \ \wedge$
$\mathsf{dom} \ (\ stDb \ ) = \mathsf{dom} \ (\ grDb \ ) \ \wedge$
$\mathsf{dom} \ (\ grDb \ ) = \mathsf{dom} \ (\ next \ ) \ \wedge$
$(\ next = \{\} \wedge head = nil \ \vee$
$(\ nil \in \mathsf{ran} \ (\ next \ ) \wedge nil \notin \mathsf{dom} \ (\ next \ ) \wedge head \in \mathsf{dom} \ (\ next \ ) \ ) \ ) \ \wedge$
$(\ next \neq \{\} \Rightarrow$
$\forall \ zz \ . \ (\ zz \in LINKS \wedge zz \in \mathsf{ran} \ (\ next \ ) \Rightarrow head \mapsto zz \in next^{*} \ ) \ ) \ \wedge$
$\forall \ link1 \ . \ (\ link1 \in \mathsf{dom} \ (\ stDb \ ) \Rightarrow abstDb \ (\ stDb \ (\ link1 \ ) \ ) = grDb \ (\ link1 \ ) \ ) \ \wedge$
$\mathsf{dom} \ (\ abstDb \ ) = \mathsf{ran} \ (\ stDb \ )$
INITIALISATION
$stDb \ , \ grDb \ , \ next \ , \ head := \{\} \ , \ \{\} \ , \ \{\} \ , \ nil$

Figure 2: The refinement machine *DbConcr*

students and their grades will be attached to the links connected in the list. The second one states that either the list is empty and *head* is equal to *nil* or *head* is in the domain of *next* and *head* is not equal to *nil* and *nil* is not in the domain but is in the range of *next*. This conjunct describes the structure of the list, i.e. the list is either empty and the head is pointing to *nil*, or the list starts from *head* and is terminated by *nil*. The third conjunct states that the list must always be properly connected, i.e. starting from the head, it should always be possible to reach the terminating nil. This is expressed by stipulating that any tuple such that its first element is *head* and its second element is any one belonging to the range of *next* must belong to the reflexive transitive closure of the function *next*.

Finally, the third part of the invariant represents a so-called "gluing invariant" which explains how the state of the abstract machine is represented in terms of the variables of its refinement. In our case it suffices to state that for all links in the domain of *stDb*, the grade recorded in *abstDb* (in the machine *DbAbstr*) for the student associated with a link in *stDb* (in the machine *DbConcr*) is equal to the grade associated with this link in *grDb* (in the machine *DbConcr*). It is also necessary to add that for all records in the abstract database there is a link in the concrete one. We achieve this by stating that the domain of *abstDb* is equal to the range of *stDb*.

As follows from the name of the following section, the variables are initialized in it. All functions are assigned empty sets and the head is assigned *nil* .

```
OPERATIONS
  append( st , gr ) ≙
    PRE
      st ∈ STUDENTS ∧ gr ∈ GRADES ∧ st ∉ ran ( stDb )
    THEN
      ANY  ll  WHERE  ll ∈ LINKS − dom ( next ) − { nil }
      THEN
        IF  next = {}  THEN
          head := ll  ||
          next := { ll ↦ nil }  ||
          stDb := { ll ↦ st }  ||
          grDb := { ll ↦ gr }
        ELSE
          stDb(ll) := st  ||
          grDb(ll) := gr  ||
          ANY  xx , next1  WHERE
            xx ∈ dom ( next ) ∧ xx ↦ nil ∈ next ∧
            next1 ∈ LINKS ⤖ LINKS ∧
            ∀ yy . ( yy ∈ LINKS ∧ yy ∈ dom ( next ) − { xx } ⇒ next1 ( yy ) = next ( yy ) ) ∧
            next1 ( xx ) = ll ∧
            next1 ( ll ) = nil
          THEN
            next := next1
          END
        END
      END
    END
  END
```

Figure 3: The refinement machine *DbConcr* (continued)

On the concrete level, definitions of operations become more elaborate. Preconditions of the operations can only be logically weakened, and they can be expressed on the variables of this refinement machine. Consider the refined **append** operation. First, we create a temporary logical variable *ll* which represents a new link to be inserted into the list *next*. This variable is assigned a value that is arbitrarily chosen from *LINKS*, is not equal to *nil*, and is a fresh value, i.e. it is not in the domain of *next*.

When appending a new student/grade record to the linked list, there can be two distinct cases, when initially the list is empty and when it is not. In the first case, we assign to *next* a tuple $ll \mapsto nil$, thus making *next* represent a list with one element *ll*, terminated by *nil*. We also make *head* to point to *ll* and associate a supplied student and grade with the link *ll*. If the linked list is not empty, we associate the supplied student and grade with the new link *ll*. After this, we create two temporary variables *xx* and *next1*, where *xx* is assigned to refer to the last element in the list before *nil* and *next1* is a copy of *next* in all the links except for the one *xx* is pointed at. In *next1*, *xx* is pointing not to *nil*, but to the new link *ll*, which,

in turn, points to *nil*. In fact, *next1* describes a new state of the function *next*. Thus the definition of the operation *append* concludes with the assignment of this new value *next1* to *next*.

For a reader well familiar with the style of B specifications, the specification presented above may appear to be somewhat convoluted, as it is quite easy to significantly shorten the definition of the refined **append** . The style of the specification presented above is motivated by the restrictions of the Alloy specification notation. We discuss these restrictions in the concluding section, as well as the modifications that it would be necessary to make to Alloy in order to permit for more natural specifications in B.

The refinement machine *DbConcr* presented in Fig.2 and Fig.3 appears to be correct, i.e. the definition of the operation *append* is consistent with respect to the invariant of the refinement, and also *append* appears to be a proper refinement of its counterpart in *DbAbcst*. But is it really correct? To be able to verify this conjecture in Alloy, we first need to consider how we can formalize the machine *DbConcr* in Alloy.

3.2 Translating the Student Grades Database to Alloy

Consider the Alloy specification presented in Fig.4. In the section `domain`, we declare three domain sets with familiar names: `STUDENTS`, `GRADES`, and `LINKS`. The keyword `fixed` is used to indicate that the marked set is unchangeable, remaining invariable before and after all operations. The next section contains the declaration of state variables. Unlike in AMN, the Alloy variable declaration not only lists the variables, but also describes their type, and partially introduces an invariant. For instance, `stDb` is declared as a partial injective function from `LINKS` to `STUDENTS`. The arrow `->` is used for constructing general relation types.

To constraint a variable to be a relation of a particular kind, such as an injective function, the domain and the range of the relation can be restricted using the so-called multiplicity characters. In the case of `stDb`, the multiplicity character used is `?` which, when attached to the name of the set in the variable declaration, makes it to have zero or one element. As `?` is attached to both the domain and the range of `stDb` signifying that for each element in the domain of `stDb` there is at most one element in its range and the other way around, i.e. `stDb` is injective.

In this specification, we also use the multiplicity character `!`, which makes a set to have exactly one element. More information on multiplicity characters and the Alloy specification notation in general can be found in [6].

In Alloy, domain-valued variables are modelled as subsets of domains rather than elements of domains, and relational image rather than function application is used to apply relations to values. Unique values are represented by singleton sets.

A declaration of the kind `domStDb : LINKS` declares `domStDb` to be a subset of the domain `LINKS`. The operator `:` is used in Alloy to indicate a subset relation while declaring a variable, and the operator `in` is used for this purpose in other parts of the specification. The variable `domStDb : LINKS` serves an auxiliary purpose only, as the machine *DbConcr* does not have a counterpart for it. This variable is necessary because Alloy does not have a function *dom* which would return a domain of a given relation. To circumvent this problem of Alloy, we have to declare the variable `domStDb` and constrain it using the definition

```
def domStDb { domStDb = {l : LINKS | some l.stDb}}
```

which makes `domStDb` to be equal to the set of such links whose image of `stDb` is non-empty. Note the usage of the operator dot (`.`), which is used for taking an image of a set through a relation.

```
model DbConcr {
      domain { fixed STUDENTS,  fixed GRADES, fixed LINKS}
      state {
            stDb : LINKS? -> STUDENTS?
            domStDb : LINKS
            ranStDb : STUDENTS
            grDb : LINKS -> GRADES?
            domGrDb : LINKS

            next : LINKS? -> LINKS?
            head : LINKS!
            domNext : LINKS
            ranNext : LINKS
            nil : fixed LINKS!

            next1 : LINKS? -> LINKS?
            domNext1 : LINKS
            ranNext1 : LINKS
      }

      def domStDb { domStDb = {l : LINKS | some l.stDb}}
      def ranStDb {ranStDb = {st : STUDENTS | some st.~stDb}}
      def domGrDb {domGrDb = {l : LINKS | some l.grDb}}
      def domNext {domNext = {l : LINKS | some l.next}}
      def ranNext {ranNext = {l : LINKS | some l.~next}}

      def domNext1 {domNext1 = {l : LINKS | some l.next1}
      def ranNext1 {ranNext1 = {l : LINKS | some l.~next1}}

      cond emptyList {all l : LINKS | no l.next}

      inv StateInv {
      domStDb = domGrDb && domGrDb = domNext

      ( emptyList  && head = nil ||
            ((nil in ranNext) && !(nil in domNext) && (head in domNext)) )

      ( !emptyList ->
            (all zz : LINKS | zz in ranNext -> zz in head.*next) )
      }

      op append(st : STUDENTS!, gr : GRADES!) {
      !(st in ranStDb)
      some ll : LINKS - domNext - nil |
            (emptyList  -> head' = ll && ll.next' = nil && ll.stDb' = st && ll.grDb' = gr &&
                  (all l : LINKS - ll | no l.next' && no l.stDb' && no l.grDb' )) &&
            ( !emptyList ->
                  ll.stDb' = st &&
                  ll.grDb' = gr &&
                  some xx : domNext | xx.next = nil &&
                        (all yy : LINKS | yy : (domNext - xx) -> yy.next1 = yy.next) &&
                        xx.next1 = ll && ll.next1 = nil &&
                  (all l : LINKS | l.next' = l.next1) &&
            (all l : LINKS - ll | l.stDb' = l.stDb && l.grDb' =l.grDb) && head' = head )
      }
}
```

Figure 4: The Alloy representation of *DbConcr*

An Alloy term `l.stDb` is equivalent to a B term $stDb(l)$.[1] The auxiliary variable `ranStDb` represents the range of the function `stDb` and is defined similarly to `domStDb`. In the definition of `ranStDb` note the usage of the `~` operator, which takes the inverse of the function. As `stDb` is defined as an injective function, its reverse is a function as well. The other variables whose name starts with `dom` or `ran` represent, respectively, domains or ranges of the corresponding functions and are all defined in a similar manner.

The variable `grDb` is represented as a partial function, while `next` is a partial injective function. There is also a declaration of the variable `head`, which is a one element set, and a variable `nil` which is marked with the keyword `fixed` turning it into a constant.

The state of the variables can be further constrained using any number of named invariants. In our case, we have only one invariant `StateInv`, which is, in fact, a translation of the invariant of the machine *DbConcr*, apart from the typing conjuncts. As Alloy prohibits comparisons of structured sets and has no predefined constant for an empty set, we had to introduce a condition `emptyList`, which in B terms is $next = \{\}$. In Alloy, `*next` represents the reflexive transitive closure of the function `next`. At this point a careful reader could have noticed that the "gluing" part of the *DbConcr* invariant does not have a counterpart in `StateInv`. As Alloy does not support the notion of refinement directly, the invariant of an Alloy model can only refer to the variables defined in this model, while the gluing invariant refers to the variables of *DbConcr* as well. The gluing invariant is of no significance for verifying consistency of the concrete *append* which is the topic of the next section. However, it is crucial for verifying the correctness of a refinement step. We discuss how to specify a gluing invariant in Alloy in Section 4.2.

The definition of the operation `append` in Alloy is, practically, a straightforward translation of its B counterpart. Alloy does not have programming language statements like "**IF THEN ELSE**", neither does it have an assignment statement. Instead, an operation in Alloy must be described as a relation between initial (unprimed) and resulting (primed) states of the variables. A B specification is built on an assumption that only the variables explicitly modified in the specification change, and all the other variables remain unchanged. In Alloy, however, it is necessary to explicitly mention that all the variables that were not modified in the definition of an operation remain in the initial state.

As was already mentioned, it is impossible (at the moment) to compare structured sets in Alloy. Thus, we cannot say $next := \{ll \mapsto nil\}$, but we should say that the image of `ll` through `next` is equal to `nil`, or `ll.next' = nil`. The definition of the operation *append* in the refinement machine *DbConcr* is formulated using a temporary variable `next1`. As in Alloy it is impossible to quantify over relations, we had to introduce this temporary variable in the state declaration. As the only invariant binding *next1* is the one making it an injective partial function from `LINKS` to `LINKS`, this is the same as stating that there exists some `next1` in the definition of the operation.

At the moment the translation from B to Alloy is done by hand. However, undoubtedly, the translation between AMN and the Alloy specification notation could be made automatic if Alloy were extended with several features. We will discuss these features in the concluding section of the paper.

4 Verifying Properties in Alloy

Let us now return to the question of whether the specification of the method *append* is correct. First, we take a look at operation consistency, and then consider the correctness of a refinement step.

[1] Should `stDb` be a general relation, the Alloy term `l.stDb` would translate into $stDb[\{l\}]$ in B.

```
Analyzing append vs. StateInv ...
Scopes: GRADES(3), LINKS(3), STUDENTS(3)
Conversion time: 10 seconds
Solver time: 13 seconds
Counterexample found:
Domains:
  LINKS = {nil,L0,L1}
Sets:
  domNext = {L0}
  domNext1 = {nil,L0,L1}
  domNext' = {nil,L0,L1}
Relations:
  next = {L0 -> nil}
  next1 = {nil -> L0, L0 -> L1, L1 -> nil}
  next' = {nil -> L0, L0 -> L1, L1 -> nil}
Skolem constants:
  ll = L1
```

Figure 5: The counter example for the operation *append*

4.1 Verifying Operation Local Consistency

If we submit `append` along with `StateInv` to the Alloy constraint analyser and indicate that the domains should be instantiated with only three elements, the analyser generates the counter example presented in Fig.4[2].

The counter example clearly violates the invariant, since after execution of the operation, `domNext'` contains `nil`, which contradicts one of the conjuncts in the invariant. Returning to the specification of `append`, it is fairly easy to spot the error. The part of the specification which deals with the case when the list is not empty describes what should be the value of the list `next1` at all the links in the domain of `next` and also at the new link `ll` we have added. This condition does not exclude, however, that `next1` can have other links. Thus, the Alloy constraint analyser is free to introduce `nil` into the domain of `next1`, which violates `StateInv`. To fix the problem, we additionally need to state that the list `next1` should only be larger than `next` by one element `ll`:

```
domNext1 = domNext + ll
```

Indeed, this amendment is sufficient to resolve the problem.

This problem can be traced back to our B specification. Therefore, all attempts to prove some of the proof obligations dealing with the consistency of the refined definition of *Append* would be futile. Now the developer, equipped with the confidence reinforced by the fact that the Alloy constraint analyser cannot find any counter examples, can return to proving the subgoals dealing with the consistency of the operation.

It is also possible to check the consistency of an operation in a different manner. Instead of translating the definition of a B operation into Alloy, it is sufficient to translate the proof obligations generated by the B tool as Alloy assertions and run the Alloy constraint analyser on them similarly to verifying operation refinement as described in the next section.

[2]We have only left the values of the relevant variables for clarity

4.2 Verifying Operation Refinement

The definition of an operation in a refinement machine can be consistent with respect to the local invariant, i.e. the part of the invariant referring only to the variables of the refined machine. However, at the same time the relation between it and its abstract counterpart can be other than refinement. Some of the proof obligations generated by the tool during verification are directed at establishing that abstract and concrete definitions of operations are, in fact, in the refinement relation. We propose to translate such proof obligations into Alloy named assertions in order to check that these proof obligations are indeed provable. Alloy assertions are the logical predicates expressed using the variables of an Alloy specification that are supposed to evaluate to true in any state the variables can be in. Accordingly, the tool attempts to find a state invalidating the predicate in the assertion.

The debugging process that we propose is then as follows. The counter example generated by the analyser can hint at modifications that must be made either to the invariant of the refinement or to the definition of an operation in B. The developer then should make these modifications to the B specification, regenerate the proof obligations, run an automated theorem prover on them, and in case any are left, translate the remaining to Alloy as assertions and repeat the debugging cycle again until the Alloy constraint analyser is unable to generate a counter example in a reasonably large scope. To become one hundred per cent certain that the refinement machine is, in fact, in the refinement relation with its abstract counterpart, the developer can then go on and prove the remaining proof obligations using an interactive theorem prover.

There is, however, a complication. As we have already mentioned, the Alloy specification notation does not provide any support for defining abstract specifications and their refinements separately. In order to express the "gluing" part of the *DbConcr*'s invariant, we have to combine all the definitions of abstract state and the definitions of its concrete implementation in the same model. Therefore, we should extend our model with the definitions for the partial function `abstDb` and its domain `domAbstDb`. The last one is defined similarly to all the other definitions of domains of functions.

```
abstDb : STUDENTS -> GRADES?
domAbstDb : STUDENTS
```

We should also extend the invariant `StateInv` to include the "gluing" conjuncts:

```
all link1 : domStDb | link1.stDb.abstDb = link1.grDb
all st : STUDENTS | some st.~stDb <-> some st.abstDb
```

To demonstrate our approach to verifying refinement, let us now return to our example. To demonstrate our approach to verification, we first need to introduce an error in the definition of *DbConcr*'s *append* that would not invalidate the consistency of the operation with respect to the invariant of the refinement machine, yet would break the refinement relation.

In the B method, the refinement machine can only be proved to be in a refinement relation with its abstract counterpart if all operations of the refinement machine preserve the gluing invariant. In our example, it states that for all links in the domain of *stDb*, the grade recorded in *abstDb* (in the machine *DbAbstr*) for the student associated with a link in *stDb* (in the machine *DbConcr*) is equal to the grade associated with this link in *grDb* (in the machine *DbConcr*). It is also states that the domain of *abstDb* is equal to the range of *stDb*. Obviously, this invariant would be violated, should we erroneously associate the submitted student not with the submitted grade but with some other *wrong* grade in *append* of *DbConcr* (see Fig.6).

Naturally, we would need to introduce the constant *wrong* in the clause **CONSTANTS** of the machine and give its type in the clause **PROPERTIES**. If we now subject the refinement

$append(\ st\ ,\ gr\ ) \ \widehat{=}$

PRE

$st \in STUDENTS \wedge gr \in GRADES \wedge st \notin \mathsf{ran}\ (\ stDb\ )$

THEN

ANY ll **WHERE** $ll \in LINKS - \mathsf{dom}\ (\ next\ ) - \{\ nil\ \}$

THEN

IF $next = \{\}$ **THEN**

$head := ll \ \ \|$

$next := \{\ ll \mapsto nil\ \} \ \ \|$

$stDb := \{\ ll \mapsto st\ \} \ \ \|$

$grDb := \{\ ll \mapsto wrong\ \}$

ELSE

... continuation as in Fig.2

Figure 6: A fragment of the erroneous definition of the operation *append* invalidating the refinement relation

machine to the standard steps of type checking, proof obligation generation, and automated theorem proving, we will be left with several proof obligations, of which "append.22" is of particular interest (see Fig.7).

The proof obligation "append.22" effectively states that the gluing invariant must hold after the execution of *append*. It must hold under the assumptions that are extracted from the **PROPERTIES** and **INVARIANT** clauses of the *DbAbtr* and *DbConcr* machines and also from the precondition of the *append* operation of this machines and the local information available from the definition of *append* in *DbConcr*.

To verify such a proof obligation in Alloy, we can represent it as a named assertion. When submitted to the constraint analyser, the latter tries to verify whether the predicate in the assertion is true in all states restricted by all invariants of the model. Therefore, while translating a B proof obligation to Alloy, we can omit all those conjuncts on the left hand side of the implication that are repeating the **INVARIANT**s and **PROPERTIES** of the abstract and concrete machines already represented in the state declaration and the invariants of the Alloy model. The obligation "append.22" can be translated as an Alloy assertion, as presented in Fig.8.

Unfortunately, at the moment the Alloy specification notation is not sufficiently rich to always permit a one-to-one translation of B. Alloy does not permit to use set operations such as intersection, union, etc. on structured sets (i.e. relations). Neither it is possible to compare structured sets. In a way, in Alloy it is impossible to state that "a certain relation is such and such", it is only possible to state "a certain relation satisfies these properties", and these "properties" should always be expressed elementwise. Therefore, to express our proof obligation in Alloy, we have to perform a case analysis on the domains of the functions participating in the right hand side of the goal.

The constraint analyser easily finds a counter example demonstrating that the assertion `P022` is not always true, i.e. that the submitted grade `gr` is not always equal to the constant `wrong`. If the developer now reverses the definition of *append* operation to its state before we introduced the "*wrong*" error and goes through the entire proposed debugging cycle, then the Alloy constraint analyser will be unable to find a counter example for the corresponding

go(append.22)

"'Component properties"" $\wedge$

...

"'Previous components properties'" $\wedge$

...

"'Previous components invariants'" $\wedge$

...

"'Component Invariant'" $\wedge$

...

"'append preconditions in previous components'" $\wedge$

...

"'append preconditions in this component'" $\wedge$

$st \notin \mathsf{ran}\ (\ stDb\ ) \wedge$

"'Local hypotheses'" $\wedge$

$ll \in LINKS\ \wedge ll \notin \mathsf{dom}\ (\ next\ ) \wedge ll \neq nil \wedge$

$next \neq \{\} \wedge xx \in \mathsf{dom}\ (\ next\ ) \wedge xx \mapsto nil \in next\ \wedge$

$next1 \in LINKS \nrightarrow LINKS \wedge next1^{-1} \in LINKS \nrightarrow LINKS \wedge$

$\mathsf{dom}\ (\ next1\ ) = \mathsf{dom}\ (\ next\ ) \cup \{\ ll\ \} \wedge$

$\forall\ yy\ .\ (\ yy \in LINKS \wedge yy \in \mathsf{dom}\ (\ next\ ) - \{\ xx\ \} \Rightarrow next1\ (\ yy\ ) = next\ (\ yy\ )\ ) \wedge$

$next1\ (\ xx\ ) = ll \wedge next1\ (\ ll\ ) = nil \wedge$

$link1 \in \mathsf{dom}\ (\ stDb \lhd\!\!\!- \{\ ll \mapsto st\ \})\ \wedge$

"'Check that the invariant (!link1.(link1: dom(stDb) ⇒ abstDb(stDb(link1)) = grDb(link1)))
is preserved by the operation - ref 4.4, 5.5'"

$\Rightarrow$

$(abstDb \cup \{st \mapsto gr\})\ (\ (\ stDb \lhd\!\!\!- \{\ ll \mapsto st\ \})\ (\ link1\ )\ ) = (\ grDb \lhd\!\!\!- \{\ ll \mapsto wrong\ \})\ (\ link1\ )$

Figure 7: The proof obligation "append.22"

assertion in a sizable scope.

5 Conclusions

As was already mentioned, the translation from B to the Alloy specification notation is done by hand, at the moment. To allow for the automatic translation, the Alloy specification language has to be extended with several features. Of these features, the ability to work with relations as with sets of tuples appears to be the most important. This should include all possible operations available for manipulating ordinary sets, such as set comparison, set union, set difference, etc. In the absence of this feature, not only the specifications are much longer, but also it is impossible to directly express properties of updated relations. The last shortcoming of Alloy is quite apparent in our translation of the proof obligation *append.22*. An introduction of the usual functions *dom* and *ran* for taking domain and range of a relation, as well as a constant {} would significantly simplify the resulting Alloy specifications, as it would be possible then, for instance, to describe the domain of a constructed function. Finally, the absence of integers (or, in fact, of any finite subset of natural numbers) and arithmetic is

```
assert PO22 {
      all st : STUDENTS, gr : GRADES, ll : LINKS, xx : LINKS, link1 : LINKS |
         !(st in ranStDb) &&
         !(st in domAbstDb) &&
         !(ll in domNext) &&
         ll != nil &&
         ! emptyList &&
         xx in domNext &&
         xx.next = nil &&
         domNext1 = domNext + ll &&
         (all yy : LINKS | yy : domNext && yy !=xx  -> yy.next1 = yy.next) &&
         xx.next1 = ll &&
         ll.next1 = nil &&
         link1 in domStDb + ll ->
            (link1 in (domStDb - ll) -> (link1.stDb in domAbstDb ->
               (link1 in (domGrDb - ll) -> link1.stDb.abstDb = link1.grDb))) &&
            (link1 in (domStDb - ll) -> (link1.stDb in domAbstDb ->
               (link1 in ll -> link1.stDb.abstDb = wrong))) &&
            (link1 in (domStDb - ll) -> (link1.stDb in st ->
               (link1 in (domGrDb - ll) -> gr = link1.grDb))) &&
            (link1 in (domStDb - ll) -> (link1.stDb in st ->
               (link1 in ll -> gr = wrong))) &&
            (link1 in ll -> (st in domAbstDb ->
               (link1 in (domGrDb - ll) -> st.abstDb = link1.grDb))) &&
            (link1 in ll -> (st in domAbstDb -> (link1 in ll -> st.abstDb = wrong))) &&
            (link1 in ll -> (st in st -> (link1 in (domGrDb - ll) -> gr = link1.grDb))) &&
            (link1 in ll -> (st in st -> (link1 in ll -> gr = wrong)))
}
```

Figure 8: The proof obligation "append.22" translated to Alloy

a very severe restriction of the current Alloy implementation, making it inapplicable to the majority of practical cases.

In principle, we perceive two major ways in which the described approach to verification can be implemented as a tool. The first way is to add Alloy-like features into the tools supporting the B method. At the moment, tools supporting the B method are supplied as integrated sets of utilities for type checking, proof obligation generating, specification animation, and theorem proving. Naturally, a utility permitting for model checking the generated proof obligations would integrate nicely with such tools. In practice, it is often infeasible to adhere to a completely formal development, as theorem proving is a very tedious and lengthy process employing highly qualified personnel. Therefore, the B method is often applied in a so-called "soft" manner, that is some of the steps of the method are omitted or validated only informally. For instance, developers might decide to informally review the remaining proof obligations which the automated theorem prover did not manage to resolve. Of course, this approach can compromise the correctness of the resulting system as it is rather easy to overlook an error. In this respect, should a B tool support a model checker similar to Alloy, it would help significantly to avoid errors and, in a way, make such an application of the B method "harder". Obviously, however, verifying proof obligations with a model checker should not discourage the developers from trying to prove the remaining proof obligations interactively. In fact, from the theoretical standpoint, even if a model checker would permit to verify a property on finite subsets of infinite domains, to make certain that the property holds on the entire domain theorem proving must be used.

The second way of implementing the suggested approach to verification as a tool is to add B-

like features to the Alloy constraint analyser. In particular, Alloy can be extended to permit for verifying refinement. Doing this, would include extending the Alloy specification language with special notation for specifying abstract and refined models. The Alloy constraint analyser could be made to incorporate a verification condition generator. Such an extension would open an entirely new scope of potential applications for Alloy.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] B-Core (UK) Limited, Oxon, UK. *B-Toolkit, On-line manual.*, 1999. Available at http://www.b-core.com/ONLINEDOC/Contents.html.

[3] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement- FDR2 user manual*, Octobre 1997. Available at www.formal.demon.co.uk/fdr2manual/index.html.

[4] M. Gordon. Introduction to the HOL system. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors, *Proceedigns of the International Workshop on the HOL Theorem Proving System and its Applications*, pages 2–3, Los Alamitos, CA, USA, Aug. 1992. IEEE Computer Society Press.

[5] G. J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[6] D. Jackson. Alloy: A lightweight object modelling notation. MIT Lab for Computer Science, July 2000.

[7] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa : the alloy constraint analyser. In *Proc. International Conference on Software Engineering*, Limerick, Ireland, June 2000.

[8] K. MacMillan. *The SMV Language*. Cadence Berkeley Labs, 1999.

[9] N. Shankar and J. M. Rushby. *PVS Tutorial*. Computer Science Laboratory, SRI International, Menlo Park, CA, Feb. 1993. Also appears in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.

[10] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1987.

[11] Steria, Aix-en-Provence, France. *Atelier B, User and Reference Manuals*, 1996. Available at http://www.atelierb.societe.com/index_uk.html.

[12] M. Woodman and B. Heal. *Introduction to VDM*. McGraw-Hill, 1993. ISBN 0-07-707434-3.