Taking *Alloy* to the Movies

Marcelo F. Frias[1,⋆], Carlos G. López Pombo[2], Gabriel A. Baum[3], Nazareno M. Aguirre[4,⋆⋆], and Tom Maibaum[5]

[1] Department of Computer Science, School of Exact and Natural Sciences, University of Buenos Aires, Argentina, and CONICET, mfrias@dc.uba.ar
[2] Department of Computer Science, School of Exact and Natural Sciences, University of Buenos Aires, Argentina, clpombo@dc.uba.ar
[3] LIFIA, School of Informatics, National University of La Plata, Argentina, and CONICET, gbaum@sol.info.unlp.edu.ar
[4] Department of Computer Science, King's College, United Kingdom, aguirre@dcs.kcl.ac.uk
[5] Department of Computer Science, King's College, United Kingdom, tom@dcs.kcl.ac.uk

Abstract. We present a modified semantics and an extension of the *Alloy* specification language. The results presented in this paper are:
(a) We show how the modified semantics of *Alloy* allows us to avoid the higher-order quantification currently used both in the composition of operations and in specifications, keeping the language first-order.
(b) We show how the extended language, which includes features from dynamic logic, enables a cleaner (with respect to previous papers) treatment of properties of executions.
(c) We show that the automatic analysis currently available for *Alloy* specifications can be fully applied in the analysis of specifications under the new semantics.
(d) We present a calculus for the extended language that is complete with respect to the extended semantics. This allows us to complement the analysis currently provided in *Alloy* with theorem proving.
(e) Finally, we show how to use the theorem prover *PVS* in order to verify *Alloy* specifications.

1 Introduction

The specification of software systems is an activity considered worthwhile in most modern development processes. In non-formal settings, specification is usually

⋆ Research partially funded by Antorchas foundation and project UBACYT X094.
⋆⋆ Currently on leave from Department of Computer Science, Universidad Nacional de Río Cuarto, Argentina.

K. Araki, S. Gnesi, and D. Mandrioli (Eds.): FME 2003, LNCS 2805, pp. 678–697, 2003.

referred to as *modelling*, since specifications allow us to build abstract models of the intended systems. Since these models are used as a means of communication with users and developers, as well as for analysis of the specified systems, it is generally considered important for modelling languages to possess a precise semantics.

Widely-used modelling languages, such as the *UML* [2] are being endowed with a formal semantics [3, 4]. Other languages, such as *VDM* [13], *Z* [23] and *Alloy* [12] were born formal, and their acceptance by software engineers greatly depends on their simplicity and usability. *Alloy* has its roots in the *Z* formal specification language. Its few constructs and simple semantics are the result of including some valuable features of *Z* and some constructs that are ubiquitous in less formal notations. This is done while avoiding to incorporate other features that would increase *Alloy*'s complexity more than necessary. *Alloy* is defined on top of what is called *relational logic*, a logic with a clear semantics based on relations. This logic provides a powerful yet simple formalism for interpreting *Alloy* modelling constructs. The simplicity of both the relational logic and the language as a whole makes *Alloy* suitable for automatic analysis. This automatic analysis is carried out using the *Alloy Analyzer* [11], a tool that incorporates state-of-the-art SAT solvers in order to search for counterexamples of specifications. *Alloy* has been used to model and analyze a number of problems of different domains, as for instance to simplify a model of the query interface mechanism of Microsoft's COM [10].

In this paper we present a modified version of *Alloy* that provides the following features:

1. The possibility of specifying functions that *formally* change the state, allowing one to describe the action that composite functions perform on models. This is possible due to the dynamic logic extension of relational logic that we will introduce. Note that, in the current version of *Alloy*, change of state is represented through the convention that some variables (e.g., primed variables) represent the final state (after execution) in function definitions. Therefore, specifications in *Alloy* provide "pictures" of a model. That is why we claim to be moving from static "pictures" to dynamic "movies".
2. The need for the second-order quantifiers in *Alloy* (see for instance [8, Section 2.4.4]) is eliminated, while keeping the expressive power and simplicity of the language. This is achieved by replacing *Alloy*'s relational logic by a similar but better-suited logic of binary relations. This logic can be automatically analyzed using the tools already available for *Alloy*.
3. An alternative technique for proving properties of executions is proposed. This technique does not make use of execution traces incorporated *within* model specifications as proposed in [12, Section 2.6], which is, to our understanding, an *ad-hoc* solution that confuses two clearly separated levels of description. Instead, our technique uses the fact that a first-order dynamic logic extending the (alternative) relational logic can be defined. This allows one to perform reasoning regarding execution traces in a simpler and more elegant way, which leads to a cleaner separation of concerns.

4. The modified version of *Alloy*'s semantics has a complete (and relatively small) proof calculus that we present here. This allows us to complement the techniques for finding counterexamples available in current *Alloy*, with theorem proving.
5. By encoding the newly defined semantics for *Alloy* in higher-order logic, we show how to verify *Alloy* specifications using the theorem prover *PVS*.

2 The *Alloy* Specification Language

We introduce the *Alloy* specification language by means of an example extracted from [12] that shows the standard features of the language. It will also help us to illustrate the shortcomings we wish to overcome.

We want to specify a memory system with cache. We start by indicating the existence of sets (of atoms) for data and addresses, which in *Alloy* are specified using signatures:

sig *Addr* { } sig *Data* { }

These are basic signatures, for which we do not assume any property of their structure. We can now say that a memory consists of set of addresses, and a (total) mapping from these addresses to data values:

```
sig Memory {
    addrs: set Addr
    map: addrs ->! Data
}
```

The "!" sign indicates that "map" is functional and total (i.e., for each element a of addrs, there exists exactly one element d in *Data* such that $\text{map}(a) = d$). Signatures defined as subsets of the set denoted by certain "parent" signature can be characterised using *signature extension*. The following signatures are defined as extensions of *Memory*:

```
sig MainMemory extends Memory {}

sig Cache extends Memory {
    dirty: set addrs
}
```

MainMemory and *Cache* are special kinds of memories. In caches, a subset of addrs is recognized as *dirty*. We can express now that a system consists of a main memory and a cache:

```
sig System {
    cache: Cache
    main: MainMemory
}
```

$problem$ ::= decl*form
decl ::= $var : typexpr$
$typexpr$ ::=
$type$
| $type \to type$
| $type \Rightarrow typexpr$

form ::=
expr *in* expr (subset)
| !form (neg)
| form && form (conj)
| form || form (disj)
| *all* $v : type$/form (univ)
| *some* $v : type$/form (exist)

expr ::=
expr + expr (union)
| expr & expr (intersection)
| expr − expr (difference)
| ∼ expr (transpose)
| expr.expr (navigation)
| +expr (closure)
| {$v : t$/form} (set former)
| Var

Var ::=
var (variable)
| $Var[var]$ (application)

M : form $\to$ $env \to Boolean$
X : expr $\to$ $env \to value$
$env = (var + type) \to value$
$value = (atom \times \cdots \times atom) +$
$(atom \to value)$

$M[a\ in\ b]e = X[a]e \subseteq X[b]e$
$M[!F]e = \neg M[F]e$
$M[F\&\&G]e = M[F]e \wedge M[G]e$
$M[F\ ||\ G]e = M[F]e \vee M[G]e$
$M[all\ v : t/F] =$
$\bigwedge\{M[F](e \oplus v \mapsto \{\ x\ \})/x \in e(t)\}$
$M[some\ v : t/F] =$
$\bigvee\{M[F](e \oplus v \mapsto \{\ x\ \})/x \in e(t)\}$

$X[a + b]e = X[a]e \cup X[b]e$
$X[a\&b]e = X[a]e \cap X[b]e$
$X[a - b]e = X[a]e \setminus X[b]e$
$X[\sim a]e = (X[a]e)^{\smile}$
$X[a.b]e = X[a]e; X[b]e$
$X[+a]e =$ the smallest r such that
$r; r \subseteq r$ and $X[a]e \subseteq r$
$X[\{v : t/F\}]e =$
$\{x \in e(t)/M[F](e \oplus v \mapsto \{\ x\ \})\}$
$X[v]e = e(v)$
$X[a[v]]e = \{\langle unit, y\rangle\ /$
$\exists x.\ \langle x, y\rangle \in e(a) \wedge \langle unit, x\rangle \in e(v)\}$

Fig. 1. Grammar and semantics of *Alloy*

As can be seen from the previous definitions, signatures define data domains and their structures. The attributes of a signature denote *relations*. For instance, the attribute "addrs" in *Memory* represents a relation from memory atoms to sets of atoms from *Addr*. Given a set (not necessarily a singleton) of *Memory* atoms m, m.addrs denotes the relational image of m under the relation denoted by addrs. This relational view of the dot notation leads to a simple and elegant semantics for dot, coherent with its intuitive navigational reading. In Fig. 1 we present the grammar and semantics of *Alloy*'s kernel. Notice that as an important difference with the previous version of *Alloy* presented in [9] where expressions range over binary relations, expressions now range over relations of arbitrary rank. Although composition of binary relations is well understood, we define composition of relations of higher rank by:

$$R;S = \{\langle a_1, \ldots, a_{i-1}, b_2, \ldots, b_j\rangle : \exists b\,(\langle a_1, \ldots, a_{i-1}, b\rangle \in R\ \wedge\ \langle b, b_2, \ldots, b_j\rangle \in S)\}\ .$$

2.1 Operations in a Model

Following the style of *Z* specifications, operations can be defined as expressions relating states from the state space described by the signature definitions. Primed variables are used to denote the resulting values, although this is a convention that is not reflected in the semantics. Consider, for instance, an operation that specifies the writing of a value to an address in a memory:

```
fun Write(m, m': Memory, d: Data, a: Addr) {
    m'.map = m.map ++ (a -> d) }
```

This definition can be easily understood, having in mind that m' is meant to denote the memory (or memory state) resulting of the function application, that a -> d denotes the pair $\langle a, d \rangle$, and ++ denotes relational override.

Consider the following more complex function definition:

```
fun SysWrite(s, s': System, d: Data, a: Addr) {
    Write(s.cache, s'.cache, d, a)
    s'.cache.dirty = s.cache.dirty + a
    s'.main = s.main }
```

There are two important points that this function definition illustrates. First, function SysWrite is defined in terms of the more primitive Write. Second, the use of Write takes advantage of the *hierarchy* defined by signature extension: function Write was defined for memories, and in SysWrite it is being "applied" to cache memories.

As explained in [12], an operation that *flushes* lines from a cache to the corresponding memory is necessary, since usually caches are small. A nondeterministic operation that flushes information from the cache to main memory is specified in the following way:

```
fun Flush(s, s': System) {
    some x: set s.cache.addrs {
        s'.cache.map = s.cache.map − { x->Data }
        s'.cache.dirty = s.cache.dirty − x
        s'.main.map = s.main.map ++
          {a: x, d: Data | d = s.cache.map[a]} }
}
```

Function Flush will serve us in Section 4.2 to illustrate one of the main problems that we try to solve. In the third line of the definition of function Flush, x->Data denotes all the pairs whose domain falls in the set x, and that range on the domain Data.

Functions can also be used to characterise *special* states. For instance, we can characterise those states in which the cache lines not marked as dirty are consistent with main memory:

```
fun DirtyInv(s: System) {
    all a : !s.cache.dirty | s.cache.map[a] = s.main.map[a] }
```

The "!" sign denotes negation, indicating in the above formula that "a" ranges over atoms that are non-dirty addresses.

2.2 Properties of a Model

As the reader might expect, a model can be enhanced by adding properties to it. These properties are written as logical formulae, much in the style of the Object

Constraint Language [16]. Properties or constraints are defined as *facts*. To give an idea of how constraints or properties are specified, we reproduce some here. We need to say that the sets of main memories and cache memories are disjoint:

$$\text{fact } \{\text{no } (\mathit{MainMemory} \ \& \ \mathit{Cache})\}$$

The expression "no x" indicates that x has no elements, and & denotes intersection. Another constraint, inherent to our specific model, states that in every system the addresses of its cache are a subset of the addresses of its main memory:

$$\text{fact } \{\text{all s: System} \mid \text{s.cache.addrs in s.main.addrs}\}$$

More complex facts can be expressed by using the significant expressive power of the relational logic.

2.3 Assertions

Assertions are the *intended* properties of a given model. Consider the following simple assertion in *Alloy*:

```
assert {
    all s: System | DirtyInv(s) && no s.cache.dirty
      => s.cache.map in s.main.map }
```

This assertion states that if "DirtyInv" holds in system "s", and there are no dirty addresses in the cache, then the cache agrees in all its addresses with the main memory. Assertions are used to test specifications. Using the *Alloy analyzer* it is possible to search for counterexamples of given assertions.

3 Features and Deficiencies of *Alloy*

Alloy is a formal specification language. What distinguishes *Alloy* from other specification languages, such as *Z* [23] or *VDM* [13], is that it has been designed with the goal of making specifications automatically analyzable. Some of its current features are:

- Fulfilling the goal of an analyzable language kept *Alloy* a simple language with an almost trivial semantics.
- *Alloy* incorporates some common idioms from object modelling. This makes *Alloy* a suitable replacement for the Object Constraint Language (OCL) [16]. The well-defined and concise syntax of *Alloy* is much easier to understand than the OCL grammar presented in [16]. A similar reasoning applies with respect to the OCL semantics. The attempt to describe all the various constructs of object modelling led to a cumbersome, incomplete, and sometimes even inconsistent semantics [1].

– The syntax of *Alloy*, which includes both a textual and graphical notation, is based on a small kernel with few constructs. Besides, the relational semantics of the kernel allows one to refer with the same simplicity to relations, sets and individual atoms.

Having described some of the features of *Alloy*, we will now describe the perceived deficiencies that will be addressed in this paper.

– Sequencing of operations, or even specifications as the one for function Flush (see Section 2.1), may require higher-order formulas. About this, Jackson says [9, Section 6.2]:

> "Sequencing of operations presents more of a language design challenge than a tractability problem. Following *Z*, one could take the formula $op1;op2$ to be short for
>
> $$some\ s : state/op1(pre, s)\ and\ op2(s, post)$$
>
> but this calls for a second-order quantifier."

  For composition of operations the problem was solved in [12] with the introduction of signatures. Signatures allow them to objectify the state and view objects containing relation attributes as atoms. However, higher-order quantifiers are used also in specifications. For instance, the definition of function Flush uses a higher-order quantifier over 1-ary relations (sets). In Section 4 we will endow the kernel of *Alloy* with a new semantics that will make higher-order quantifiers unnecessary.
– In [12], Jackson et al. present a methodology for proving properties of executions. The method consists of the introduction of a new sort of *finite traces*. Each element in a trace stands for a state in an execution. In this context, proving that a given assertion is invariant under the execution of some operations is reduced to proving the validity of the assertion in the last element of every finite trace. Even though from a formal point of view the technique is correct, from the modelling point of view it seems less appropriate. When a software engineer writes an assertion, verifying the assertion should not demand a modelling effort. In order to keep an adequate separation of concerns between the modelling stage and the verification stage, verifying the assertion should reduce to proving a property in a suitable logic. The logic extending *Alloy* that we propose in Section 5 will enable us to verify this kind of assertions (i.e., assertions regarding executions) in a simple and elegant way.
– *Alloy* was designed with the goal of being automatically analyzable, and thus theorem proving was not considered a critical issue. Nevertheless, having the possibility of combining model checking with theorem proving as in the STeP tool [15] is a definite improvement. Providing *Alloy* with theorem proving is not trivial, since *Alloy*'s relational logic does not admit a complete proof calculus. Despite this fact, in Section 6 we present a complete deductive system for an alternative logic *extending Alloy*'s kernel.

4 A New Semantics for *Alloy*

In most papers the semantics of *Alloy*'s kernel is defined in terms of binary relations. The current semantics [12] is given in terms of relations of arbitrary finite arity. The modified semantics for *Alloy* that we will present goes back to binary relations. This was our choice for the following three main reasons:

1. *Alloy*'s kernel operations such as transposition or transitive closure are only defined on binary relations.
2. There exists a complete calculus for reasoning about binary relations with certain operations (to be presented next).
3. It is possible (and we will show how) to deal with relations of rank higher than 2 within the framework of binary relations we will use.

4.1 Fork Algebras

Fork algebras [5] are described through few equational axioms. The intended models of these axioms are structures called *proper fork algebras*, in which the domain is a set of binary relations (on some base set, let us say B), closed under the following operations for sets:

- *union* of two binary relations, denoted by $\cup$,
- *intersection* of two binary relations, denoted by $\cap$,
- *complement* of a binary relation, denoted, for a binary relation r, by $\overline{r}$,
- the *empty* binary relation, which does not relate any pair of objects, and is denoted by $\emptyset$,
- the *universal* binary relation, namely, $B \times B$, that will be denoted by 1.

Besides the previous operations for sets, the domain has to be closed under the following operations for binary relations:

- *transposition* of a binary relation. This operation swaps elements in the pairs of a binary relation. Given a binary relation r, its transposition is denoted by $\breve{r}$,
- *composition* of two binary relations, which, for binary relations r and s is denoted by $r;s$,
- *reflexive–transitive closure*, which, for a binary relation r, is denoted by r^*,
- the *identity* relation, denoted by Id.

Finally, a binary operation called *fork* is included, which requires the base set B to be closed under an injective function $\star$. This means that there are elements x in B that are the result of applying the function $\star$ to elements y and z. Since $\star$ is injective, x can be seen as an encoding of the pair $\langle y, z\rangle$. The application of fork to binary relations R and S is denoted by $R \nabla S$, and its definition is given by: $R \nabla S = \{\, \langle a, b \star c\rangle : \langle a, b\rangle \in R \text{ and } \langle a, c\rangle \in S \,\}$.

Once the class of proper fork algebras has been presented, the class of fork algebras is axiomatized with the following formulas:

1. Your favorite set of equations axiomatizing Boolean algebras. These axioms define the meaning of union, intersection, complement, the empty set and the universal relation.
2. Formulas defining composition of binary relations, transposition, reflexive–transitive closure and the identity relation:
   $x;(y;z) = (x;y);z$,
   $x;Id = Id;x = x$,
   $(x;y) \cap z = \emptyset$ iff $(z;\breve{y}) \cap x = \emptyset$ iff $(\breve{x};z) \cap y = \emptyset$,
   $x^* = Id \cup (x;x^*)$,
   $x^*;y;1 \leq (y;1) \cup \left(x^*;(\overline{y;1} \ \cap \ (x;y;1))\right)$.
3. Formulas defining the operator ∇:
   $x \nabla y = (x;(Id \nabla 1)) \cap (y;(1 \nabla Id))$,
   $(x \nabla y);(w \nabla z)^{\smile} = (x;\breve{w}) \cap (y;\breve{z})$,
   $(Id \nabla 1)^{\smile} \nabla (1 \nabla Id)^{\smile} \leq Id$.

The axioms given above define a class of models. Proper fork algebras satisfy the axioms [6], and therefore belong to this class. It could be the case that there are models for the axioms that are not proper fork algebras. Fortunately, as was proved in [6], [5, Thm. 4.2], if a model is not a proper fork algebra then it is isomorphic to one. Notice also that binary relations are first-order citizens in fork algebras, and therefore quantification over binary relations is first-order.

4.2 Fork-Algebraic Semantics of *Alloy*

In order to give semantics to *Alloy*, we will give semantics to *Alloy*'s kernel. We provide the modified (in comparison to [12]) denotational semantics in Fig. 2. This semantics is given through two meaning functions. Function N gives meaning to formulas. It requires an environment in which types and variables with free occurrences take values, and yields a boolean as a result indicating wether the formula is true or not in the environment. Similarly, function Y gives meaning to expressions. Since expressions can also contain variables, the environment is again necessary. The general assumption is that variables in the environment get as values relations in an arbitrary fork algebra $\mathfrak{A}$ whose universe we will denote by U.

Representing Objects and Sets. We will represent sets by binary relations contained in the identity relation. Thus, for an arbitrary type t and an environment env, $env(t) \subseteq Id$ must hold. That is, for a given type t, its meaning in an environment env is a binary relation contained in the identity binary relation. Similarly, for an arbitrary variable v of type t, $env(v)$ must be a relation of the form $\{ \langle x, x \rangle \}$, with $\langle x, x \rangle \in env(t)$. This is obtained by imposing the following conditions on $env(v)$[1]:

$$env(v) \subseteq env(t),$$
$$env(v);1;env(v) = env(v),$$
$$env(v) \neq \emptyset \ .$$

[1] The proof requires relation 1 to be of the form $B \times B$ for some nonempty set B.

$$
\begin{array}{l}
N : \text{form} \to env \to Boolean \\
Y : \text{expr} \to env \to U \\
env = (var + type) \to U.
\end{array}
$$

$$
\begin{array}{l}
N[a\ in\ b]e = Y[a]e \subseteq Y[b]e \\
N[!F]e = \neg N[F]e \\
N[F\&\&G]e = N[F]e \wedge N[G]e \\
N[F\ ||\ G]e = N[F]e \vee N[G]e \\
N[all\ v : t/F] = \bigwedge\{N[F](e \oplus v{\mapsto}x)/x : e(t)\} \\
N[some\ v : t/F] = \bigvee\{N[F](e \oplus v{\mapsto}x)/x : e(t)\}
\end{array}
$$

$$
\begin{array}{l}
Y[a + b]e = Y[a]e \cup Y[b]e \\
Y[a\&b]e = Y[a]e \cap Y[b]e \\
Y[a - b]e = Y[a]e \cap \overline{Y[b]e} \\
Y[\sim a]e = (Y[a]e)^{\smile} \\
Y[a.b]e = Y[a]e \bullet Y[b]e \\
Y[+a]e = Y[a]e;(Y[a]e)^{*} \\
Y[\{v : t/F\}]e = \bigcup\{x : e(t)/N[F](e \oplus v{\mapsto}x)\} \\
Y[v]e = e(v) \\
Y[a[v]]e = e(v);e(a)
\end{array}
$$

Fig. 2. The new semantics of *Alloy*

Actually, given binary relations x and y satisfying the properties:

$$y \subseteq Id, \quad x \subseteq y, \quad x;1;x = x, \quad x \neq \emptyset, \tag{1}$$

it is easy to show that x must be of the form $\{\, \langle a, a\rangle \,\}$ for some object a. Thus, given an object a, by a we will also denote the binary relation $\{\, \langle a, a\rangle \,\}$. Since y represents a set, by $x : y$ we assert the fact that x is an object of type y, which implies that x and y satisfy the formulas in (1).

Eliminating Higher-Order Quantification. We will show now that by giving semantics to *Alloy* in terms of fork algebras, higher-order quantifiers are not necessary. Recalling the specification of function Flush in Section 2.1, the specification has the shape

$$\text{some x : set t / F .} \tag{2}$$

This is recognized within *Alloy* as a higher-order formula [8]. Let us analyze what happens in the modified semantics. Since t is a type (set), it stands for a subset of Id. Similarly, subsets of t are subsets of the identity, which are contained in t. Thus, formula (2) is an abbreviation for

$$\exists x\,(x \subseteq t \ \wedge \ F)\,,$$

which is a first-order formula when x ranges over binary relations in a fork algebra.

Regarding the higher-order formulas that appear in the composition of operations, discussed in Section 3, no higher-order formulas are required in our setting. Formula

$$some\ s : state/op1(pre, s)\ and\ op2(s, post) \tag{3}$$

is first-order with the modified semantics. Operations $op1$ and $op2$ can be defined as binary predicates in a first-order language for fork algebras, and thus formula (3) is first-order.

Representing and Navigating Relations of Higher Rank in Fork Algebras. In a proper fork algebra the relations π and ρ defined by

$$\pi = (Id \nabla 1)^{\smile}, \quad \rho = (1 \nabla Id)^{\smile}$$

behave as projections with respect to the encoding of pairs induced by the injective function $\star$. Their semantics in a proper fork algebra $\mathfrak{A}$ whose binary relations range over a set B, is given by

$$\pi = \{ \langle a \star b, a \rangle : a, b \in B \}, \quad \rho = \{ \langle a \star b, b \rangle : a, b \in B \} \ .$$

Given a n-ary relation $R \subseteq A_1 \times \cdots \times A_n$, we will represent it by the binary relation $\{ \langle a_1, a_2 \star \cdots \star a_n \rangle : \langle a_1, \ldots, a_n \rangle \in R \}$. This will be an invariant in the representation of n-ary relations by binary ones.

Recalling signature *Memory*, attribute map stands in *Alloy* for a ternary relation map $\subseteq$ *Memory* $\times$ addrs $\times$ Data. In our framework it becomes a binary relation map' whose elements are pairs of the form $\langle m, a \star d \rangle$ for m : *Memory*, a : Addr and d : Data. Given an object (in the relational sense — cf. 4.2) m : Memory, the navigation of the relation map' through m should result in a binary relation contained in Addr $\times$ Data. Given a relational object $a : t$ and a binary relation R encoding a relation of rank higher than 2, we define the navigation operation $\bullet$ by

$$a \bullet R = \breve{\pi}; Ran\,(a; R)\,; \rho \ . \tag{4}$$

Operation *Ran* in (4) returns the range of a relation as a subset of the identity relation. It is defined by $Ran\,(x) = (x;1) \cap Id$. Its semantics in terms of binary relations is given by $Ran\,(R) = \{ \langle a, a \rangle : \exists b\,(\langle b, a \rangle \in R) \}$.

For a binary relation R representing a relation of rank 2, navigation is easier. Given a relational object $a : t$, we define $a \bullet R = Ran\,(a; R)$.

Going back to our example about memories, it is easy to check that for a relational object $m' : Memory$ such that $m' = \{ \langle m, m \rangle \}$,

$$m' \bullet map' = \{ \langle a, d \rangle : a \in Addr, d \in Data \ and \ \langle m, a \star d \rangle \in map' \} \ .$$

Analyzing the Modified *Alloy*. An essential feature of *Alloy* is its adequacy for automatic analysis. Thus, an immediate question is what is the impact of the modified semantics in the analysis of *Alloy* specifications. In the next paragraphs, we will argue that the new semantics can fully profit from the current analysis procedure. Notice that the *Alloy* tool is a refutation procedure. As such, if we want to check if an assertion α holds in a specification S, we must search for a model of $S \cup \{ \neg \alpha \}$. If such a model exists, then we have found a counterexample that refutes the assertion α. Of course, since first-order logic is undecidable,

this cannot be a decision procedure. Therefore, the *Alloy* tool searches for counterexamples of a bounded size, in which each set of atoms is bounded to a finite size or "*scope*".

A counterexample is an environment, and as such it provides sets for each type of atom, and values (relations) for the constants and the variables. We will show now that whenever a counterexample exists according to *Alloy*'s standard semantics, the same is true for the fork algebraic semantics.

For the next theorem we assume that whenever the transpose operation or the transitive closure occur in a term, they affect a binary relation. Notice that this is the assumption in [12]. We also assume that whenever the navigation operation is applied, the argument on the left-hand side is a unary relation (set). This is because our representation of relations of arity greater than two makes defining the generalized composition more complicated than desirable. At the same time, the use of navigation in object-oriented settings usually falls in the situation modelled by us.

Given an environment e, we define the environment e' (according to the new semantics) by:

- Given a type T, $e'(T) = \{ \langle a, a\rangle : a \in e(T) \}$.
- Given a variable v such that $e(v)$ is a n-ary relation,

$$e'(v) = \begin{cases} \{ \langle a, a\rangle : a \in e(v) \} & \text{if } n = 1, \\ \{\langle a_1, a_2 \star \cdots \star a_n\rangle : \langle a_1, a_2, \ldots, a_n\rangle \in e(V)\} & \text{otherwise.} \end{cases}$$

Theorem 1. *Given a formula α, $M[\alpha]e = N[\alpha]e'$.*

The proof of Thm. 1 is by induction on the structure of formulas. Theorem 1 shows that all the work that has been done so far in the analysis of *Alloy* specifications can be fully profitted by the newly proposed semantics. The theorem proposes a method for analyzing *Alloy* specification (according to the new semantics), as follows:

1. Give the *Alloy* specification to the current *Alloy* analyzer.
2. Get a counterexample, if any exists within the given scopes.
3. Build a counterexample for the new semantics from the one provided by the tool, The new counterexample is defined in the same way environment e' is defined from environment e above. Notice that Thm. 1 implies that a counterexample exists with respect to the standard semantics if and only if one exists for the newly provided semantics.

5 Adding Dynamic Features to *Alloy*

In this section we extend *Alloy*'s kernel syntax and semantics in a way that is fully consistent with the extension we performed in Section 4. The reason for this extension is twofold. First, we want to provide a setting in which state transformations are not just simulated by distinguishing between primed and

non-primed variables, but rather are identifiable in the semantics. Second, the framework allows one to reason about properties of executions in a simple and clean way. The section is structured as follows. In Section 5.1 we introduce the syntax and semantics of first-order dynamic logic. In Section 5.2 we present the formalism of dynamic logic over fork algebras. Finally, in Section 5.3 we show how to reason about executions.

5.1 Dynamic Logic

Dynamic logic is a formalism suitable for reasoning about programs. From a set of atomic actions (usually assignments of terms to variables), and using adequate combinators, it is possible to build complex actions. The logic then allows us to state properties of these actions, which may hold or not in a given structure. Actions can change (as usually programs do) the values of variables. We will assume that each action reads and/or modifies the value of finitely many variables. When compared with classical first-order logic, the essential difference is the dynamic content of dynamic logic, which is clear in the notion of satisfiability. While satisfiability in classical first–order logic depends on the values of variables in one valuation (state), in dynamic logic it may be necessary to consider two valuations in order to reflect the change of values of program variables; one valuation holds the values of variables *before* the action is performed, and another holds the values of variables *after* the action is executed.

Along the paper we will assume a fixed (but arbitrary) finite signature $\Sigma = \langle s, A, F, P \rangle$, where s is a sort, $A = \{ a_1, \ldots, a_k \}$ is the set of atomic action symbols, F is the set of function symbols, and P is the set of atomic predicate symbols. Atomic actions contain input and output formal parameters. These parameters are later instantiated with actual variables when actions are used in a specification.

The sets of *programs* and *formulas* on Σ are mutually defined in Fig. 3.

As is standard in dynamic logic, states are valuations of the program variables (the actual parameters for actions). The environment *env* assigns a domain $\mathbf{s}$ to sort s in which program variables take values. The set of states is denoted by ST. For each action symbol $a \in A$, *env* yields a binary relation on the set of states, that is, a subset of $ST \times ST$. The environment maps function symbols to concrete functions, and predicate symbols to relations of the corresponding arity. The semantics of the logic is given in Fig. 3.

5.2 Dynamic Logic over Fork Algebras

In order to define first-order dynamic logic over fork algebras, we always include in the set of function symbols of signature Σ the constants 0, 1, Id; the unary symbols $^-$ and $^\smile$; and the binary symbols $+$, $\cdot$, ; and ∇. Since these signatures include all operation symbols from fork algebras, they will be called *fork signatures*.

We will call theories containing the identities specifying the class of fork algebras *fork theories*. By working with fork theories we intend to describe structures

	$Q : \text{form} \to ST \to Boolean$
	$P : \text{action} \to \mathcal{P}(ST \times ST)$
action ::= $a_1, \ldots a_k$ (atomic actions)	$Z : \text{expr} \to ST \to \mathbf{s}$
| *skip*	
| action + action (nondeterministic choice)	$Q[p(t_1, \ldots, t_n)]\mu = (Z[t_1]\mu, \ldots, Z[t_n]\mu) \in env(p)$
| action ; action (sequential composition)	$Q[!F]\mu = \neg Q[F]\mu$
| action* (finite iteration)	$Q[F \&\& G]\mu = Q[F]\mu \wedge Q[G]\mu$
| dform? (test)	$Q[F \mid\mid G]\mu = Q[F]\mu \vee Q[G]\mu$
	$Q[all\ v : t\ /\ F]\mu = \bigwedge\{Q[F](\mu \oplus v{\mapsto}x)/x \in env(t)\}$
expr ::= *var*	$Q[some\ v : t\ /\ F]\mu = \bigvee\{Q[F](\mu \oplus v{\mapsto}x)/x \in env(t)\}$
| $f(\text{expr}_1, \ldots, \text{expr}_k)$ ($f \in F$ with arity k)	$Q[\ [a]F\ ]\mu = \bigwedge\{Q[F]\nu / \langle \mu, \nu \rangle \in P(a)\}$
dform ::= $p(\text{expr}_1, \ldots, \text{expr}_n)$ ($p \in P$)	$P[a] = env(a)$ (atomic action)
| !dform (negation)	$P[skip] = \{\ \langle \mu, \mu \rangle : \mu \in ST\ \}$
| dform && dform (conjunction)	$P[a + b] = P[a] \cup P[b]$
| dform || dform (disjunction)	$P[a\,;b] = P[a] \circ P[b]$
| *all* v : *type*/dform (universal)	$P[a^*] = (P[a])^*$
| *some* v : *type*/dform (existential)	$P[\alpha?] = \{\ \langle \mu, \mu \rangle : Q[\alpha]\mu\ \}$
| [action]dform (box)	
	$Z[v]\mu = \mu(v)$
	$Z[f(t_1, \ldots, t_k)]\mu = env(f)(Z[t_1]\mu, \ldots, Z[t_k]\mu)$

Fig. 3. Syntax and semantics of dynamic logic

for dynamic logic whose domains are sets of binary relations. This is indeed the case as shown in the following theorem whose proof will appear in an extended paper due to space limitations.

Theorem 2. *Let Σ be a fork signature, and Ψ be a fork theory. For each model $\mathfrak{A}$ for Ψ there exists a model $\mathfrak{B}$ for Ψ, isomorphic to $\mathfrak{A}$, in which the domain* **s** *is a set of binary relations.*

The previous theorem is essential, and its proof (which uses [5, Thm. 4.2]), heavily relies on the use of fork algebras rather than plain relation algebras [24]. A model for a fork theory Ψ is a structure satisfying all the formulas in Ψ. Such a structure can, or cannot, have binary relations in its domain. Theorem 2 shows that models whose domains are not a set of binary relations are isomorphic to models in which the domain is a set of binary relations. This allows us to look at specifications in first-order dynamic logic over fork algebras, and interpret them as properties predicating about binary relations.

Notice that fork signatures contain action symbols, function symbols (including at least the fork algebra operators), and predicate symbols. The relationship to *Alloy* is established as follows. We use actions to model *Alloy* functions. This is particularly adequate, since state modifications described by functions are better viewed as the result of performing an action on an input state. Thus, a definition of a function f of the form

$$\text{fun } f(s, s')\{\alpha(s, s')\} \tag{5}$$

has as counterpart a definition of an action f of the form

$$[s\ f\ s']\alpha(s, s')\ . \tag{6}$$

Although it may be hard to find out what are the differences between (5) and (6) just by looking at the formulas, the differences rely in the semantics,

and in the fact that actions can be sequentially composed, iterated or nondeterministically chosen, while *Alloy* functions cannot.

5.3 Specifying and Proving Properties of Executions

Suppose we want to show that a given property P is invariant under sequences of applications of the operations "Flush", and "SysWrite" from an initial state. A technique useful for proving invariance of property P consists of proving P on the initial states, and proving for every non initial state and every operation O that $P(s) \wedge O(s, s') \Rightarrow P(s')$ holds. This proof method is sound but incomplete, since the invariance may be violated in non-reachable states. Of course it would be desirable to have a proof method in which the considered states were exactly the reachable ones. This motivated in [12] the introduction of *traces* in *Alloy*.

The following example, extracted from [12], shows signatures for clock ticks and for traces of states.

```
sig Tick {}

sig SystemTrace {
    ticks: set Tick,
    first, last: Tick,
    next: (ticks - last) ! → ! (ticks - first),
    state: ticks → ! System }
```

The following "fact" states that all ticks in a trace are reachable from the first tick, that a property called "Init" holds in the first state, and finally that the passage from one state to the next is through the application of one of the operations under consideration.

```
fact {
    first.next* = ticks
    Init(first.state)
    all t: ticks - last   |
      some s = t.state, s' = t.next.state   |
        Flush (s,s')
        || some d : Data, a : Addr | SysWrite(s,s',d,a) }
```

If we now want to prove that P is invariant, it suffices to show that P holds in the final state of every trace. Notice that non reachable states are no longer a burden because all the states in a trace are reachable from the states that occur before.

Even though from a formal point of view the use of traces is correct, from a modelling perspective it is less adequate. Traces are introduced in order to cope with the lack of real state change of *Alloy*. They allow us to port the primed variables used in single operations to sequences of applications of operations.

Dynamic logic [7], on the other hand, was created in the early 70s with the intention of faithfully reflecting state change. In the following paragraphs we will

show how it can be used to specify properties of executions of *Alloy* operations. In order to increase the readability of formulas, rather than writing

$$\alpha \Rightarrow [a]\beta, \tag{7}$$

we will use the alternative notation $\{\alpha\}\ a\ \{\beta\}$. This notation is particularly adequate because a formula like formula (7) indeed asserts that action a is partially correct with respect to the pre-condition α and the post-condition β.

Going back to the example of cache systems, we will use an auxiliary predicate "Write", modelling the evolution of a memory state when main memory is written:

$$\text{Write}(\text{m}_0, \text{m} : \mathit{Memory}, \text{d} : \mathit{Data}, \text{a} : \mathit{Addr}) \iff \text{m.map} = \text{m}_0\text{.map} +\!+ (\text{a} \rightarrow \text{d})\ .$$

Then, specification of functions SysWrite and Flush is done as follows:

{ s = s_0 }

SysWrite(s: *System*)

{ some d: *Data*, a: *Addr* |
Write(s_0.cache, s.cache, d, a)
s.cache.dirty = s_0.cache.dirty + a
s.main = s_0.main }

{ s = s_0 }

Flush(s: *System*)

{ some x: set s_0.cache.addrs |
s.cache.map = s_0.cache.map - x→Data
s.cache.dirty = s_0.cache.dirty - x
s.main.map = s_0.main.map ++
{a: x, d: Data | d = s_0.cache.map[a]} }

Notice that the previous specifications are as understandable as the ones given in *Alloy*. Moreover, using dynamic logic for the specification of functions allows us to assert the invariance of a property P under finite applications of functions SysWrite and Flush as follows:

$$\mathit{Init}(s) \wedge P(s) \ \Rightarrow\ [(\mathit{SysWrite}(s) + \mathit{Flush}(s))^*]P(s)\ .$$

More generally, suppose now that we want to show that property Q is invariant under sequences of applications of arbitrary operations $O_1, \ldots, O_k$, starting from states s described by a formula *Init*. Specification of the problem in our setting is done through the formula $\mathit{Init} \wedge Q \ \Rightarrow\ [(O_1 \cup \cdots \cup O_k)^*]Q$.

As an instance of the properties of executions that can be proved in our formalism, let us consider a system whose cache agrees with main memory in all non-dirty addresses. A consistency criterion of the cache with main memory is that after finitely many executions of SysWrite or Flush, the resulting system must still satisfy invariant DirtyInv. In Section 7 we will prove this property, which is specified in the extended *Alloy* by:

$$\text{all } s : \text{System} \ /\ \text{DirtyInv}(s) => [(\text{SysWrite}(s) + \text{Flush}(s))^*]\text{DirtyInv}(s)\ . \tag{8}$$

Notice also that if after finitely many executions of SysWrite and Flush we flush all the dirty addresses in the cache to main memory, the resulting cache should fully agree with main memory. We will specify the property in this section, and leave its proof for Section 7. In order to specify this property we need to specify the function that flushes all the dirty cache addresses. The specification is as follows:

$$
\begin{array}{l}
\{\ \mathrm{s} = \mathrm{s}_0\ \} \\
\quad \mathrm{DSFlush}(\mathrm{s} : \mathit{System}) \\
\{\ \mathrm{s.cache.dirty} = \emptyset \\
\quad \mathrm{s.cache.map} = \mathrm{s}_0.\mathrm{cache.map} - \mathrm{s}_0.\mathrm{cache.map}[\mathrm{s}_0.\mathrm{cache.dirty}] \\
\quad \mathrm{s.main.map} = \mathrm{s}_0.\mathrm{main.map} +\!\!+ \mathrm{s}_0.\mathrm{cache.map}[\mathrm{s}_0.\mathrm{cache.dirty}]\ \}
\end{array}
$$

We specify the property establishing the agreement of the cache with main memory by: $\mathrm{FullyAgree}(s : \mathit{System}) \iff s.\mathrm{cache.map}\ \mathit{in}\ s.\mathrm{main.map}$.

Once "DSFlush" and "FullyAgree" have been specified, the property is specified in the extended *Alloy* by:

$$
\begin{array}{l}
\text{all } s : \mathrm{System} \ /\ \mathrm{DirtyInv}(s) => \\
\qquad\qquad [(\mathrm{SysWrite}(s) + \mathrm{Flush}(s))^*; \mathrm{DSFlush}(s)]\mathrm{FullyAgree}(s). \quad (9)
\end{array}
$$

Notice that there is no need to mention traces in the specification of the previous properties. This is because traces appear in the semantics of the Kleene star and not in the syntax, which shows an adequate separation of concerns.

6 A Complete Calculus

The set of axioms for the extended *Alloy* is the set of axioms for classical first-order logic, enriched with the axioms for fork algebras and the following formulas:

$$
\begin{array}{ll}
\langle P\rangle\alpha \wedge [P]\beta \ \Rightarrow\ \langle P\rangle(\alpha \wedge \beta), & \langle P\rangle(\alpha \vee \beta) \ \Leftrightarrow\ \langle P\rangle\alpha \vee \langle P\rangle\beta, \\
\langle P_0 + P_1\rangle\alpha \ \Leftrightarrow\ \langle P_0\rangle\alpha \vee \langle P_1\rangle\alpha, & \langle P_0; P_1\rangle\alpha \ \Leftrightarrow\ \langle P_0\rangle\langle P_1\rangle\alpha, \\
\langle \alpha? \rangle\beta \ \Leftrightarrow\ \alpha \wedge \beta, & \alpha \vee \langle P\rangle\langle P^*\rangle\alpha \ \Rightarrow\ \langle P^*\rangle\alpha, \\
\langle P^*\rangle\alpha \ \Rightarrow\ \alpha \vee \langle P^*\rangle(\neg\alpha \wedge \langle P\rangle\alpha), & \langle x \leftarrow t\rangle\alpha \ \Leftrightarrow\ \alpha[x/t], \\
\alpha \ \Leftrightarrow\ \widehat{\alpha}, &
\end{array}
$$

where $\widehat{\alpha}$ is α in which some occurrence of program P has been replaced by the program $z \leftarrow x; P'; x \leftarrow z$, for z not appearing in α, and P' is P with all the occurrences of x replaced by z.

The inference rules are those for classical first-order logic plus generalization rule for necessity, and the infinitary convergence rule:

$$
\frac{\alpha}{[P]\alpha} \qquad\qquad \frac{(\forall n : nat)(\alpha \Rightarrow [P^n]\beta)}{\alpha \Rightarrow [P^*]\beta}
$$

A proof of the completeness of the calculus is presented in [7, Thm. 15.1.4]. Joining this theorem with the completeness of the axiomatization of fork algebras [5, Thm. 4.3], it follows that the above described calculus is complete with respect to the semantics of the extended *Alloy*.

7 Verifying *Alloy* Specifications with *PVS*

As has been shown in previous sections, the extended *Alloy* is a language suitable for the description of systems behavior. There are different options in order to reason about such descriptions. Techniques such as model checking, sat solving and theorem proving give the possibility to detect systems flaws in early stages of the design lifecycle.

Regarding the problem of theorem proving, there are several theorem provers that can be used to carry out this task. *PVS* (*Prototype Verification System*), is a powerful and widely used theorem prover that has shown very good results when applied to the specification and verification of real systems [19]. Thus, we will concentrate on the use of this particular theorem prover in order to prove assertions from *Alloy* specifications.

As it has been described in the basic *PVS* bibliography [20–22], *PVS* is a theorem prover built on classical higher-order logic. The main purpose of this tool is to provide formal support during the design of systems, in a way in which concepts are described in abstract terms to allow a better level of analysis. *PVS* provides very useful mechanisms for system specification such as an advanced data-type specification language [18], the notion of subtypes and dependent types [22], the possibility to define parametric theories [22], and a collection of powerful proof commands to carry out propositional, equality, and arithmetic reasoning [20]. These proof commands can be combined to form proof strategies. The last feature simplifies the process of developing, debugging, maintaining, and presenting proofs.

Using *PVS* to reason about *Alloy* specifications is not trivial because *Alloy* is not supported by the *PVS* tool. To bridge this gap, a proof checker was built by encoding the new semantics for *Alloy* in *PVS*' language [14].

Taking as a case-study the memories with cache (systems) presented in Section 5.3, in order to build the *PVS* specification we provided *PVS* with the definition of the symbols for the language of fork algebras, the definition of the semantics of the symbols of fork algebras, the definition of the atomic actions required in the model, and the assertion to be verified in the model. In Figs. 4 and 5 we show, as examples, the *PVS* translation of formulas (8) and (9).

```
Preservation_of_DirtyInv: LEMMA
  FORALL_(v(cs), DirtyInv(v(cs)) IMPLIES
                 [](*(SysWrite(v(cs))+Flush(v(cs))), DirtyInv(v(cs))))
```

Fig. 4. *PVS* translation of Formula (8).

```
Consistency_criterion: THEOREM
  FORALL_(v(cs), DirtyInv(v(cs)) IMPLIES
                 [](*(SysWrite(v(cs))+Flush(v(cs)))//DSFlush(v(cs)),
                    FullyAgree(v(cs))))
```

Fig. 5. *PVS* translation of Formula (9).

We have proved in *PVS* the properties stated in Figs. 4 and 5. This required the implementation of new proof strategies in *PVS*.

8 Conclusions

We have presented an extension of *Alloy* that incorporates the following features:

1. Through the use of fork algebras in the semantics, quantifications that were higher-order in *Alloy* are first-order in the extension.
2. Through the extension of *Alloy* with dynamic logic, static models in which dynamic content was described using conventions such as primed variables, now have a real dynamic content.
3. The use of dynamic logic provides a clean and simple mechanism for the specification of properties of executions.
4. Combining the completeness of a calculus for dynamic logic and the complete calculus for fork algebras gives us a complete calculus for the extended *Alloy*. This enables theorem proving as an alternative to analysis by refutation.
5. Finally, we have also extended the theorem prover *PVS* in order to prove properties specified in the extended *Alloy*.

Acknowledgements

We wish to thank Daniel Jackson for reading preliminary versions of this paper and making valuable suggestions. We are also thankful to Sam Owre and Natarajan Shankar for their help in the verification of properties in *PVS*.

References

1. Bickford M. and Guaspari D., *Lightweight Analysis of UML*. TM-98-0036, Odyssey Research Associates, Ithaca, NY, November 1998.
2. Booch G., Jacobson I. and Rumbaugh J., *The Unified Modeling Language User Guide*, The Addison-Wesley Object Technology Series, 1998.
3. Evans A., Kent S. and Selic B. (eds.), *UML 2000 - The Unified Modeling Language. Advancing the Standard*, Proceedings of the Third International Conference in York, UK, October 2-6, 2000. Springer Verlag Berlin, LNCS 1939.
4. France R. and Rumpe B. (eds.), *UML '99 - The Unified Modeling Language. Beyond the Standard*, Proceedings of the Second International Conference in Fort Collins, Colorado, USA, October 28-30, 1999. Springer Verlag Berlin, LNCS 1723.

5. Frias M., *Fork Algebras in Algebra, Logic and Computer Science*, World Scientific Publishing Co., Series Advances on Logic, 2002.
6. Frias, M. F., Haeberer, A. M. and Veloso, P. A. S., *A Finite Axiomatization for Fork Algebras*, Logic Journal of the IGPL, Vol. 5, No. 3, 311–319, 1997.
7. Harel D., Kozen D. and Tiuryn J., *Dynamic Logic*, MIT Press, October 2000.
8. Jackson D., *Micromodels of Software: Lightweight Modelling and Analysis with Alloy*, 2002.
9. Jackson D., *Alloy: A Lightweight Object Modelling Notation*, ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 11, Issue 2 (April 2002), pp. 256-290.
10. Jackson D. and Sullivan K., *COM Revisited: Tool Assisted Modelling and Analysis of Software Structures*, Proc. ACM SIGSOFT Conf. Foundations of Software Engineering. San Diego, November 2000.
11. Jackson D., Schechter I. and Shlyakhter I., *Alcoa: the Alloy Constraint Analyzer*, Proceedings of the International Conference on Software Engineering, Limerick, Ireland, June 2000.
12. Jackson, D., Shlyakhter, I., and Sridharan, M., A Micromodularity Mechanism. Proc. ACM SIGSOFT Conf. Foundations of Software Engineering/European Software Engineering Conference (FSE/ESEC '01), Vienna, September 2001.
13. Jones C.B., *Systematic Software Development Using VDM*, Prentice Hall, 1995.
14. Lopez Pombo C.G., Owre S. and Shankar N., *An* $\mathbf{A_g}$ *proof checker using PVS as a semantic framework*, Technical Report SRI-CSL-02-04, SRI International, June 2002.
15. Manna Z., Anuchitanukul A., Bjorner N., Browne A., Chang E., Colon M., de Alfaro L., Devarajan H., Sipma H. and Uribe T., *STeP: The Stanford Temporal Prover*, http://theory.stanford.edu/people/zm/papers/step.ps.Z. Technical report STAN-CS-TR-94-1518, Computer Science Department, Stanford University, July 1994.
16. *Object Constraint Language Specification". Version 1.1*, 1 September 1997.
17. Owre S., Rushby J.M. and Shankar N., *PVS: A prototype verification system*, In Deepak Kapur, editor, 11th International Conference on Automated Deduction (CADE), volume 607 of Lecture Notes in Artificial Intelligence, pp. 748–752, Saratoga, NY, jun 1992. Springer-Verlag.
18. Owre S. and Shankar N., *Abstract datatypes in PVS*, Technical Report CSL-93-9R, SRI International, December 1993. Subtantially revised in June 1997.
19. Owre S., Shankar N., Rushby J,M, and Stringer-Calvert D.W.J., *PVS: An Experience Report*, in Proceedings of Applied Formal Methods—FM-Trends 98, Lecture Notes in Computer Science 1641, 1998, pp. 338–345.
20. Owre S., Shankar N., Rushby J,M, and Stringer-Calvert D.W.J., *PVS Prover Guide*, SRI International, version 2.4 edition, November 2001.
21. Owre S., Shankar N., Rushby J.M. and Stringer-Calvert D.W.J., *PVS System Guide*, SRI International, version 2.4 edition, December 2001.
22. Owre S., Shankar N., Rushby J.M. and Stringer-Calvert D.W.J., *PVS Language reference*, SRI International, version 2.4 edition, December 2001.
23. Spivey J.M., *Understanding Z: A Specification Language and Its Formal Semantics*, Cambridge Tracts in Theoretical Computer Science, 1988.
24. Tarski, A. and Givant, S.,*A Formalization of Set Theory without Variables*, A.M.S. Coll. Pub., vol. 41, 1987.