Reasoning About Static and Dynamic Properties in Alloy: A Purely Relational Approach

Marcelo F. Frias[*]
Department of Computer Science
School of Sciences
Universidad de Buenos Aires
Argentina
and
CONICET
mfrias@dc.uba.ar

Carlos G. López Pombo
Department of Computer Science
School of Sciences
Universidad de Buenos Aires
Argentina
clpombo@dc.uba.ar

Gabriel A. Baum
LIFIA - School of Informatics
Universidad Nacional de La Plata
Argentina
and
CONICET
gbaum@sol.info.unlp.edu.ar

Nazareno M. Aguirre
Department of Computer Science
FCEFQyN
Universidad Nacional de Río Cuarto
Argentina
aguirre@dc.exa.unrc.edu.ar

Thomas S. E. Maibaum
Department of Computer Science
King's College London
United Kingdom
tom@dcs.kcl.ac.uk

ABSTRACT

We study a number of restrictions associated with the first-order relational specification language *Alloy*. The main shortcomings we address are:

- the lack of a complete calculus for deduction in Alloy's underlying formalism, the so called relational logic,
- the inappropriateness of the *Alloy* language for describing (and analysing) properties regarding execution traces.

The first of these points was not regarded as an important issue during the genesis of *Alloy*, and therefore has not been taken into account in the design of the relational logic. The second point is a consequence of the static nature of *Alloy* specifications, and has been partly solved by the developers of *Alloy*; however, their proposed solution requires a complicated and unstructured characterisation of executions.

We propose to overcome the first problem by translating the relational logic to the equational calculus of the *fork algebras*. Fork algebras provide a (purely relational) formalism close to Alloy, that possesses a complete equational deductive calculus. Regarding the second problem, we propose to extend Alloy by adding *actions*. These actions, unlike Alloy functions, do modify the state. Much the same as programs in dynamic logic, actions can be sequentially composed and iterated, allowing to state properties of execution traces at an appropriate level of abstraction. Since automatic analysis is one of Alloy's main features, and this paper aims to provide a deductive calculus for Alloy,

- we show that the extension hereby proposed does not sacrifice the possibility of using SAT solving techniques for automated analysis,
- we show how to extend the calculus for the relational logic to a calculus to which the extension of Alloy with actions can be translated. This provides a complete calculus for reasoning about the extension of Alloy.

[*]Research partially funded by Antorchas foundation.

1. INTRODUCTION

The specification of software systems is an activity considered worthwhile in most modern development processes. Some *specification languages* are informal, meaning that the notations on which they are based are not precisely defined. Informal specification languages are usually referred to as *modeling* languages, since specifications allow us to build abstract models of the intended systems. Due to the lack of a precise semantics, informal specifications must usually be complemented with natural language annotations or some other mechanisms, in order not to fall into ambiguous understandings of what is being modeled. However, informal specifications are still useful as a means for communication between developers, documentation and even for performing some (restricted) kinds of analysis. The *UML* [5] is an example of a widely used informal specification language, whose specifications (based on a variety of languages) are centered on notions from object orientation.

Formal approaches to software specification, on the other hand, are those based on well defined notations, founded on solid (usually mathematical) grounds. Formal specification languages are better suited for analysis, due to their precise semantics, but they are usually more complex, and require familiarity and experience with the manipulation of mathematical definitions. So, their acceptance by software engineers greatly depends on their simplicity and usability.

There exists a wide range of formal specification languages, based on a variety of logics and other formalisms. A subset of these languages, in which we are interested, are the so called *model oriented* formal specification languages. Their approach to specification consists of describing systems by building mathematical models of them. Traditionally, model oriented specification languages describe a system by defining its state space, and its operations as state transformations. Some examples of model oriented formal specification languages are *B* [1], *VDM* [25], *Z* [38] and *Alloy* [24].

Formal semantics is not necessarily enough for making specifications analysable: effective analysis mechanisms must be defined. Furthermore, it is generally accepted that, due to the difficulties associated with the use of formal methods, appropriate (software) tool support for the analysis is a must. We are particularly interested in *Alloy*, a language designed with (fully automated) analysability of specifications as a priority, and which has recently gained increasing attention. *Alloy* has its roots in the *Z* formal specification language, and its few constructs and simple semantics are the result of putting together some valuable features of *Z* and some constructs that are normally found in informal notations. This is done while avoiding incorporation of other features that would increase *Alloy*'s complexity more than necessary.

Alloy is defined on top of what is called *relational logic* (RL), a logic with a clear semantics based on relations. This logic provides a powerful yet simple formalism for interpreting *Alloy* modeling constructs. The simplicity of both the relational logic and the language as a whole makes *Alloy* suitable for automatic analysis. The main analysis technique associated with *Alloy* is essentially a counterexample extraction mechanism, based on SAT solving. Basically, given a system specification and a statement about it, a counterexample of this statement (under the assumptions of the system description) is exhaustively searched for. Since first-order logic is not decidable (and the relational logic is a proper extension of first-order logic), SAT solving cannot be used in general to guarantee the consistency of (or, equivalently, the absence of counterexamples for) a theory; then, the exhaustive search for counterexamples has to be performed up to certain bound k in the number of elements in the universe of the interpretations. Thus, this analysis procedure can be regarded as a *validation* mechanism, rather than a *verification* procedure. Its usefulness for validation is justified by the interesting idea that, in practice, if a statement is not true, it often exists a counterexample of it of small size:

> "First-order logic is undecidable, so our analysis cannot be a decision procedure: if no model is found, the formula may still have a model in a larger scope. Nevertheless, the analysis is useful, since many formulas that have models have small ones." (cf. [19, p. 1])

The above described analysis has been implemented by the Alloy Analyzer [23], a tool that incorporates state-of-the-art SAT solvers in order to search for counterexamples of specifications. *Alloy* and its tool support have been used with some success to model and analyse a number of problems of different domains, such as, for instance, the simplification of a model of the query interface mechanism of Microsoft's COM [22].

1.1 Contributions of this Paper

The contributions of this paper are twofold. First, notice that deduction was not regarded as an important issue during the genesis of *Alloy*, and therefore has not been taken into account in the design of the relational logic. In order to overcome this difficulty, we introduce the fork relational logic (FRL), as the equational theory of fork algebras [14] extended with reflexive-transitive closure. The interpretation of *Alloy*'s underlying relational logic within FRL allows us to define a purely relational and complete equational calculus for reasoning about *Alloy* specifications. Moreover, we will show that translating *Alloy* specifications into FRL, instead of doing so into the relational logic, does not compromise the analysability of specifications. In fact, resorting to FRL enables us to analyse strictly more properties than with the Alloy Analyzer under the standard *Alloy* semantics [23].

Second, we notice that *Alloy* is inappropriate as a language for the description and analysis of properties regarding execution traces of systems. This is due to the static nature of *Alloy* specifications (a deficiency shared with other model oriented specification languages), and although has been partly solved by the developers of *Alloy*, their proposed solution requires a complicated and unstructured characterisation of executions. In order to address this problem, we propose extending *Alloy* with *actions*, and enhancing FRL's expressiveness with dynamic logic features. Functions in *Alloy*, as schemes for operations in *Z*, serve the purpose of defining *state changes*, by relating variables corresponding to the state prior to the operation's execution with variables corresponding to the state resulting from the execution of the operation. A number of conventions, such as the fact that primed variables correspond to post-execution state, are central to the correct interpretation of function definitions. We believe that actions (as will be defined in this paper), unlike functions, are better qualified for describing state change, especially for the purpose of expressing properties regarding executions.

We will argue that, as a result, the newly defined *dynamic* Alloy (DynAlloy) is better suited (at least when compared to [24, Section 2.6]) for modeling execution of operations, and reasoning about execution traces. Even a SAT solving based analysis, similar to that defined for standard *Alloy*, can be provided in order to *validate* properties regarding execution traces.

Since one of the intended contributions of this paper is providing a complete calculus for Alloy, it is worthwhile asking whether the deductive calculus for the relational logic presented in this paper can be extended to DynAlloy. We show that extending FRL to a dynamic logic over fork algebras (denoted by FDL), we again obtain a purely relational and complete proof calculus; this enables us to perform deductive reasoning also about properties of executions specified in DynAlloy.

An interesting side effect of adopting FRL (and its extension FDL) as our foundation is that there is no need for second-order quantifiers in the composition of *Alloy* functions (see for instance [20, Section 2.4.4]). Also, the availability of encodings for the semantics of FRL and its extension FDL into higher-order logic allows us to verify *Alloy* specifications (even those involving actions and executions) using a higher-order logic theorem prover, such as *PVS* [34].

The relationships among the formalisms involved in our approach are depicted in Fig. 1.

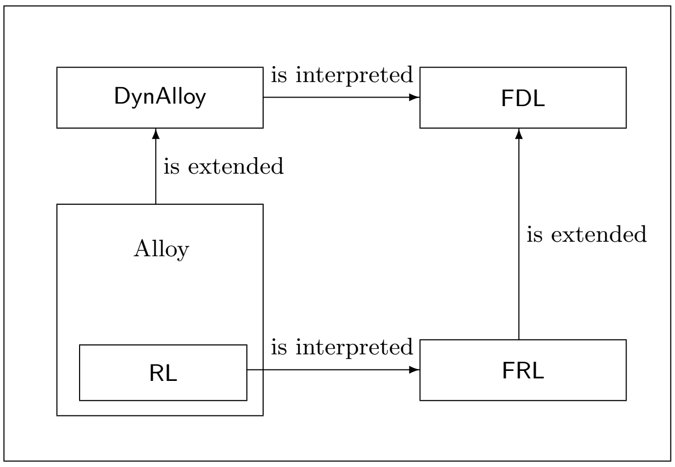


Figure 1: Relationships among the formalisms.

1.2 Related Work

As we mentioned before, deduction was not considered an important issue when *Alloy* was created; instead, the design of the language was centered on the idea of providing efficient automated analysis of specifications via SAT solving. In the case of *Alloy*, SAT solving based analysis provides a *validation* mechanism for specification. Theorem proving, on the other hand, would provide a mechanism for the *verification* of properties.

To the best of our knowledge, since the beginnings of *Alloy*, not much work has been done regarding the use of theorem proving in the analysis of *Alloy* specifications. We are aware of the work of Arkoudas et al. [3] on their tool Prioni, presented some time after the first submission of this paper at RelMiCS'03[1]. Prioni uses a semi automatic theorem prover called Athena [2] in order to prove properties regarding *Alloy* specifications. Arkoudas et al.'s characterisation of *Alloy* allows one to reason about specifications, using (semi automatic) theorem proving. However, the proposed characterisation does not capture well some features of *Alloy* and its relational logic, such as, for instance, the uniform treatment for scalars and singletons. Quoting the authors,

> "Recall that all values in Alloy are relations. In particular, Alloy blurs the type distinction between scalars and singletons. In our Athena formalization, however, this distinction is explicitly present and can be onerous for the Alloy user." (cf. [3, p. 6])

Prioni has also a number of further shortcomings, such as the, in our opinion, awkward representations of *Alloy*'s composition operation '$\cdot$' and of ordered pairs [3, p. 4]. This tool, however, manages to integrate the use of theorem proving with the SAT solving based analysis of the Alloy Analyzer, cleverly assisting theorem proving with SAT solving and vice versa. The mechanisms used to combine SAT solving and theorem proving are independent of the theorem prover used and the axiomatic characterisation of *Alloy*. Thus, they could also be employed to combine the Alloy Analyzer with other approaches to reasoning about *Alloy* specifications, such as, for instance, the one presented in this paper.

The deduction system we propose is based on purely relational (i.e., only the sort of relations is present) calculi for FRL and FDL, which are complete with respect to the semantics of the corresponding logics. These logics allow for a uniform treatment of scalars and singletons, and thanks to their expressiveness, the characterisation of the constructs of *Alloy* and RL is straightforward. We believe that 'minimal mathematics [for the end user]', which comes from adopting well understood concepts such as sets and relations and is one of the motivations of *Alloy*, is not lost by using these logics to enhance *Alloy*.

There is a wide range of (semi-)automatic techniques for software verification and validation. A particularly successful branch is that of *model checking* [7]. By model checking we mean the well known approach of representing a (finite state) program as a *model* in a certain (often modal) logic, and then checking whether that model satisfies or not a logical property. There exist various approaches to efficient model checking, such as the automata-theoretic [40], the semantic [8] and the symbolic ones [29].

Alloy, as well as our approach to deduction, is not directly related to model checking, since the subject of verification (or validation) is not, in principle, a transition system (i.e., the representation of the possible execution traces of a program); instead, *Alloy*'s target is in the description and analysis of structural properties of systems [21]. Nevertheless, model checking techniques might be useful for the verification of properties regarding traces of executions of *Alloy* specifications. In fact, as demonstrated in [24], there exists an interest in describing and analysing the possible behaviors of systems specified in *Alloy*; furthermore, as we said, it is one of our aims to contribute to a better characterisation of these behaviors. The description of executions proposed in [24], by introducing system traces, clock ticks, etc, *as part of* the model of the system, obscures the differences between what is the description of the system itself and what is part of the machinery necessary for "talking about executions". We believe that this unstructured merge of the system description and the characterisation of behaviors would complicate the possibility of applying model checking techniques to verify properties of executions. Our approach,

[1]Note that our approach is previous to Arkoudas et al.'s: [3] includes a reference to a the preliminary version of this paper [16].

on the other hand, clearly separates the two different levels of specification, leaving the description of executions to the upper layer dynamic logic. Model checking would then be more easily applicable.

We restrict ourselves to the study of a well organised and simple characterisation of executions of *Alloy* specifications. The exact difficulties related to the use of model checking techniques for the analysis of properties regarding executions of *Alloy* specifications are beyond the scope of this paper.

1.3 Structure of the Paper

The remainder of this paper is organised as follows. In Section 2, we present a description of the syntax and semantics of the current version of standard *Alloy*, as presented in [24]. In Section 3, we present the main features of *Alloy* and some of its shortcomings; we also discuss some, in our opinion, desirable improvements to the language. In Section 4, we introduce the fork relational logic FRL, and present a semantics preserving mapping from RL to FRL that allows for deduction of RL properties in FRL. In Section 5 we extend *Alloy* to DynAlloy by adding features from dynamic logic, and show how properties of executions can be represented in DynAlloy. We also show how to analyze DynAlloy using the Alloy Analyzer. In Section 6 we present a complete deductive calculus for DynAlloy. In Section 7 we present an extension of the theorem prover *PVS* in order to verify FDL specifications. Finally, in Section 8 we present our conclusions and proposals for further work.

2. THE ALLOY SPECIFICATION LANGUAGE

In this section, we introduce the reader to the *Alloy* specification language, by means of an example extracted from [24]. This example serves as a means for illustrating the standard features of the language and their associated semantics, and will also help us demonstrate the shortcomings we wish to overcome.

Suppose we want to specify a systems involving memories with cache. We might recognise that, in order to specify memories, datatypes for data and addresses are especially necessary. We can then start by indicating the existence of disjoint sets (of atoms) for data and addresses, which in *Alloy* are specified using signatures:

```
sig Addr { }          sig Data { }
```

These are basic signatures. We do not assume any special properties regarding the structures of data and addresses.

With data and addresses already defined, we can now specify what constitutes a memory. A possible way of defining memories is by saying 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 symbol "!" in the above definition indicates that "map" is functional and total (for each element a of addrs, there exists exactly one element d in *Data* such that map$(a) = d$).

Alloy allows for the definition of signatures as subsets of the set denoted by other "parent" signature. This is done via what is called *signature extension.* For the example, one could define other (perhaps more complex) kinds of memories as extensions of the *Memory* signature:

```
sig MainMemory extends Memory {}

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

With these definitions, *MainMemory* and *Cache* are special kinds of memories. In caches, a subset of addrs is recognised as *dirty.*

A system might now be defined to be composed of a main memory and a cache:

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

As the previous definitions show, signatures are used to define data domains and their structure. The attributes of a signature denote *relations.* For instance, the "addrs" attribute in signature *Memory* represents a binary relation, from memory atoms to sets of atoms from *Addr.* Given a set m (not necessarily a singleton) of *Memory* atoms, m.addrs denotes the relational image of m under the relation denoted by addrs. This leads to a relational view of the dot notation, which is simple and elegant, and preserves the intuitive navigational reading of dot, as in object orientation. Signature extension, as we mentioned before, is interpreted as inclusion of the set of atoms of the extending signature into the set of atoms of the extended signature.

In Fig. 2, we present the grammar and semantics of *Alloy*'s relational logic. An important difference with respect to the previous version of *Alloy*, as presented in [21], is that expressions now range over relations of arbitrary rank, instead of being restricted to binary relations. Composition of binary relations is well understood; but for relations of higher rank, the following definition for the composition of relations has to be considered:

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

Operations for transitive closure and transposition are only defined for binary relations. Thus, function X in Fig. 2 is partial.

2.1 Operations in a Model

So far, we have just shown how the structure of data domains can be specified in *Alloy*. Of course, one would like to be able to define operations over the defined domains. Following the style of Z specifications, operations in *Alloy* 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 just a convention, not reflected in the semantics.

In order to illustrate the definition of operations in *Alloy*, 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)
}
```

$\mathit{problem}$ ::= decl*form
decl ::= $\mathit{var} : \mathit{typexpr}$
$\mathit{typexpr}$ ::=
type
| $\mathit{type} \rightarrow \mathit{type}$
| $\mathit{type} \Rightarrow \mathit{typexpr}$

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

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

Var ::=
var (variable)
| $\mathit{Var}[\mathit{var}]$ (application)

$M :$ form $\rightarrow \mathit{env} \rightarrow \mathit{Boolean}$
$X :$ expr $\rightarrow \mathit{env} \rightarrow \mathit{value}$
$\mathit{env} = (\mathit{var} + \mathit{type}) \rightarrow \mathit{value}$
$\mathit{value} = (\mathit{atom} \times \cdots \times \mathit{atom}) +$
$(\mathit{atom} \rightarrow \mathit{value})$

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

$X[a + b]e = X[a]e \cup X[b]e$
$X[a\&b]e = X[a]e \cap X[b]e$
$X[a - b]e = X[a]e \setminus X[b]e$
$X[\sim a]e = \{ \langle x, y \rangle : \langle y, x \rangle \in X[a]e \}$
$X[a.b]e = X[a]e ; X[b]e$
$X[+a]e =$ the smallest r such that
$r ; r \subseteq r$ and $X[a]e \subseteq r$
$X[\{v : t/F\}]e =$
$\{x \in e(t)/M[F](e \oplus v \mapsto \{ x \})\}$
$X[v]e = e(v)$
$X[a[v]]e = \{\langle y_1, \ldots, y_n \rangle /$
$\exists x. \langle x, y_1, \ldots, y_n \rangle \in e(a) \wedge \langle x \rangle \in e(v)\}$

Figure 2: Grammar and semantics of Alloy

The intended meaning of this definition can be easily understood, having in mind that m' is meant to denote the memory (or memory state) resulting of the application of function Write, a -> d denotes the ordered pair $\langle$a, d$\rangle$, and ++ denotes relational overriding, defined by[2]

$$R{+}{+}S = \{ \langle a_1, \ldots, a_n \rangle : \langle a_1, \ldots, a_n \rangle \in R \ \wedge \ a_1 \notin \mathsf{dom}\,(S) \} \cup S \ .$$

We have already seen a number of constructs available in *Alloy*, such as the dot notation and signature extension, that resemble object oriented definitions. Operations, however, represented by functions in *Alloy*, are not "attached" to signature definitions, as in traditional object-oriented approaches. Instead, functions describe operations of the whole set of signatures, i.e. the model. So, there is no notion similar to that of class, as a mechanism for encapsulating data (attributes) and behavior (operations or methods).

In order to illustrate a couple of further points, 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 issues exhibited in this function definition. 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: note that function Write was defined for memories, and in SysWrite it is being "applied" to cache memories.

As explained in [24], an operation that *flushes* lines from a cache to the corresponding memory is necessary in order to have a realistic model of memories with cache, since usually caches are smaller than main memories. A (nondeterministic) operation that flushes information from the cache to main memory can be 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]}
    }
}
```

In the third line of the above definition of function Flush, x->Data denotes all the ordered pairs whose domains fall into the set x, and that range over the domain Data. Function Flush will be used in Section 4.1.5 to illustrate one of the main problems that we try to solve.

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

```
fun DirtyInv(s: System) {
    all a : !s.cache.dirty |                                  (1)
            s.cache.map[a] = s.main.map[a] }
```

In this context, the symbol "!" 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 (axioms) to it. These properties are written as logical formulas, much in the style of the Object Constraint Language [31]. Properties or constraints in *Alloy* are defined as *facts*. To give an idea of how constraints or properties are specified, we reproduce some here. It might be necessary to say that the sets of main memories and cache memories are disjoint:

fact {no (*MainMemory* & *Cache*)}

In the above expression, "no x" indicates that x has no elements, and & denotes intersection. Another important constraint inherent to the presented model is that, in every system, the addresses of its cache are a subset of the addresses of its main memory:

fact {all s: System | s.cache.addrs in s.main.addrs}

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

2.3 Assertions

Assertions are the *intended* properties of a given model. Consider, for instance, the following simple *Alloy* assertion, regarding the presented example:

```
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 check specifications. Using the Alloy analyzer, it is possible to validate assertions, by searching for possible counterexamples for them, under the constraints imposed in the specification of the system.

[2]Given a n-ary relation R, $\mathsf{dom}\,(R)$ denotes the set $\{ a_1 : \exists a_2, \ldots, a_n$ such that $\langle a_1, a_2, \ldots, a_n \rangle \in R \}$.

3. FEATURES AND DEFICIENCIES OF ALLOY

In this section, we summarise what are, to our understanding, the main features and deficiencies of the *Alloy* language.

Alloy is a formal specification language which has, as any other formal specification language, a formal syntax and semantics. Contrary to the approach of most model oriented formal specification languages, such as *Z* [38], *VDM* [25] or *B* [1], *Alloy*'s semantics is strongly based on the use of *relations*. A main distinguishing characteristic of *Alloy*, that we mentioned before in this paper, is that it has been designed with the goal of making specifications automatically analysable. This restriction forced the developers of *Alloy* to keep the language simple, not including even simple data types such as integers, floats, rationals or lists.

Alloy has evolved significantly since its origins. Although the language is rather simple, it is surprisingly expressive, especially useful for the description of the structure of systems and their properties. Some of the important features of *Alloy*'s current version, as described in [21], are the following:

- Fulfilling the goal of an analysable language made *Alloy* a simple language, with a clear and elegant semantics based on relations.
- Regardless of its simplicity, *Alloy* supports some constructs which resemble common idioms of object modeling. Perhaps this feature is one of the main reason why *Alloy* reaches a broader audience than that of some other formal specification languages. Also thanks to this characteristic of the language, *Alloy* can be regarded as a suitable alternative for the Object Constraint Language (OCL) [31]. The well defined and concise syntax of *Alloy* is much easier to understand than the, in our opinion, rather cumbersome OCL grammar presented in [31]. A similar argument applies when comparing *Alloy* and OCL with respect to their semantics. OCL's attempt to describe the various constructs of object modeling led to a cumbersome, incomplete, and perhaps even inconsistent semantics [4].
- The syntax of *Alloy*, which includes both a textual and a graphical notations, is based on a small underlying formalism, RL, with few constructs. The relational semantics of RL allows one to refer with the same simplicity to relations, sets and individual atoms.

Alloy also has some, to our understanding, important deficiencies. As we have explained, we are interested in addressing two main drawbacks of *Alloy*; these are the following:

- *Alloy* was designed with the goal of making specifications automatically analysable by means of SAT solving based techniques. Theorem proving was not then considered a critical issue in the design of the language and its underlying foundations.

  Fully automatic techniques have limitations. In the case of *Alloy*, SAT solving based analysis allows one to *validate* a property of a specification, but we cannot use the analysis for proper *verification*. There is some evidence of the fact that (semi automated) deduction can be used successfully, especially in combination with fully automatic analysis. The Stanford Temporal Prover (STeP) [28], for instance, is a good example of a tool combining, with great success, fully automatic verification (in this case, model checking) with semi automated deduction.

  Providing *Alloy* with theorem proving is not a particularly complicated task. As we mentioned, Arkoudas et al. [3] have even implemented a tool for theorem proving in *Alloy*. However, their calculus resorts to the set-theoretical definition of Alloy's operators, thus loosing the purely relational flavor of Alloy. Actually, it is not clear whether a complete, purely relational calculus for Alloy even exists. Completeness is an advantageous feature, because it expresses the fact that one has all the deductive power one might need; in other words, all the statements (expressible in the logic) which are consequences of the *axioms* of a specification are provable, if one counts on a complete proof system for the logic.

  Despite the fact that a complete proof calculus for RL has not yet been found, we present in Section 6 a complete deductive system for FRL, a logic *extending Alloy*'s foundational formalism RL.

- Whereas *Alloy* makes a great choice for describing structural properties of systems, the language is, in our opinion, inappropriate for the description of properties regarding behaviors of systems. This is due to a particularity of *Alloy*, inherited from *Z*: specifications are descriptions of the static aspects of systems, such as structural invariants and the like, but one has no direct way of expressing facts regarding execution traces.

  In [24], Jackson et al. present a methodology for checking properties of executions in *Alloy*. The method presented in [24] consists of the representation, together with the static description of a system, of its execution traces. It involves incorporating into the model of a system elements such as a sort for its *finite traces*, operations for clock ticks, first and last points in a trace, etc. In this context, checking if a given assertion is invariant under the execution of some operations is reduced to checking for the validity of the assertion in the last element of every finite trace. Since the model of execution traces is incorporated as part of the system description, SAT solving based analysis is still applicable.

  Although this approach is sound, we believe it is not the best way of tackling *Alloy*'s limitations with respect to the description of behaviors. This is because of, essentially, two reasons: first, when a software engineer writes an assertion, validating the assertion should not demand additional modeling efforts; second, in order to keep an appropriate separation of concerns in the modeling activity, the static and dynamic parts of system description should be clearly identifiable.

  Our proposal in order to overcome this problem is presented in Section 5. It consists of extending *Alloy* to a more expressive specification language, called *dynamic* Alloy (DynAlloy), which separates the static and dynamic aspects of a specification in a simple and better organised manner. DynAlloy supports the description of assertions regarding executions. DynAlloy can then be interpreted over a dynamic logic extending FRL. A SAT solving based analysis, similar to that defined for standard *Alloy*, can be provided in order to *validate* properties regarding execution traces in DynAlloy. Also, the dynamic logic over FRL admits a complete proof system. Therefore, we are also able to do theorem proving regarding properties of executions.

- In the definition of some necessary elements of a system specification, such as sequencing of operations, or even specifications as the one for function Flush (see Section 2.1), one may require higher-order formulas. Quoting *Alloy*'s developers:

  > "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." (cf. [21, Section 6.2])

  A partial solution to this problem was proposed in [24], consisting of a treatment for operation composition via the use of signatures. However, higher-order quantifiers are still used within specifications. For instance, the definition of function Flush uses a higher-order quantifier over unary relations (sets).

  Our approach, combining the fork-algebraic logic and its dynamic logic extension, has as a side effect the elimination of the need for higher-order quantification.

4. A COMPLETE EQUATIONAL CALCULUS FOR ALLOY, BASED ON FORK ALGEBRAS

In most papers the semantics of Alloy's relational logic is defined in terms of binary relations. The current semantics [24] is given in terms of relations of arbitrary finite arity. The formalism FRL that we will present goes back to binary relations. This was our choice for the following three main reasons:

1. Alloys relational logic 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 Closure Fork Algebras

Fork algebras [14] 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:

- the *identity* relation (on B), denoted by Id.
- *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 $r;s$.

Finally, a binary operation called *fork* is included, which requires the base set B to be closed under an injective function $\star : B \times B \to B$. 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:

$$R \nabla S = \{\, \langle a, b \star c\rangle : \langle a, b\rangle \in R \text{ and } \langle a, c\rangle \in S \,\} \ .$$

Closure fork algebras are then obtained from fork algebras by adding *reflexive–transitive closure*, which, for a binary relation r, is denoted by r^*.

Once the class of proper closure fork algebras has been presented, it is axiomatized with the following formulas and inference rules. These give raise to the *Fork Relational Logic*, that we will denote by FRL.

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 $(\breve{x};z) \cap y = \emptyset$.
3. Formulas defining the operator ∇:
   $x \nabla y = (x;(Id \nabla 1)) \cap (y;(1 \nabla Id))$,
   $(x \nabla y);(w \nabla z)\breve{} = (x;\breve{w}) \cap (y;\breve{z})$,
   $(Id \nabla 1)\breve{} \nabla (1 \nabla Id)\breve{} \le Id$.
4. Formulas defining reflexive-transitive closure:
   $x^* = Id \cup (x;x^*)$,
   $x^*;y;1 \le (y;1) \cup \left(x^*;(\overline{y;1} \ \cap\ (x;y;1))\right)$.

The inference rules for the calculus are those for equational logic (see for instance [6, p. 94]), plus the following equational (but infinitary) proof rule for reflexive-transitive closure[3]:

$$\frac{\vdash 1' \le y \qquad x^i \le y \vdash x^{i+1} \le y}{\vdash x^* \le y}$$

[3]Given $i > 0$, by x^i we denote the relation inductively defined as follows: $x^1 = x$, and $x^{i+1} = x;x^i$.

The axioms and rules given above define a class of models. Proper closure fork algebras satisfy the axioms [13], and therefore belong to this class. It could be the case that there are models for the axioms that are not proper closure fork algebras. Fortunately, as was proved in [15] (which heavily relies on [13]), if a model is not a proper closure 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.

In Section 4.1.3 we will need to handle fork terms involving variables denoting relations. Following the definition of the semantics of Alloy, we define a mapping Y that, given an environment in which these variables receive values, homomorphically allows to calculate the values of terms. We also present a mapping that allows us to assign semantics to fork algebraic equations. The definitions are given in Fig. 3. The set U is the domain of a proper fork algebra, and therefore a set of binary relations.

$$\begin{aligned}
&Y : \text{expr} \to env \to U\\
&N : \text{expr} \times \text{expr} \to env \to Boolean\\
&env = (var + type) \to U.\\[1em]
&Y[\emptyset]e = \text{smallest element in } U\\
&Y[1]e = \text{largest element in } U\\
&Y[\overline{a}]e = \overline{Y[a]e}\\
&Y[a \cup b]e = Y[a]e \cup Y[b]e\\
&Y[a \cap b]e = Y[a]e \cap Y[b]e\\
&Y[\breve{a}]e = (Y[a]e)^{\smile}\\
&Y[Id]e = Id\\
&Y[a;b]e = Y[a]e;Y[b]e\\
&Y[a \nabla b]e = Y[a]e \nabla Y[b]e\\
&Y[a^*]e = (Y[a]e)^*\\
&Y[v]e = e(v)\\[1em]
&N[t_1,t_2]e = (Y[t_1]e = Y[t_2]e)
\end{aligned}$$

Figure 3: Semantics of fork terms and equations involving variables.

4.1.1 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 $env(v)$[4]:

$$\begin{aligned}
&env(v) \subseteq env(t),\\
&env(v);1;env(v) = env(v),\\
&env(v) \neq \emptyset\ .
\end{aligned}$$

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

$$y \subseteq Id, \quad x \subseteq y, \quad x;1;x = x, \quad x \neq \emptyset, \tag{2}$$

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 (2).

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

4.1.2 Representing and Navigating Relations of Higher Rank in Fork Algebras

In a proper fork algebra the relations π and ρ defined by

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

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

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

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

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.

From fork, π and ρ we can define a new binary operator called *cross* (and denoted by $\otimes$) by

$$R \otimes S = (\pi;R)\ \nabla\ (\rho;S)\ .$$

From a set-theoretical point of view, cross can be understood as follows

$$R \otimes S = \{ \langle a \star b, c \star d \rangle : \langle a,c \rangle \in R\ \wedge\ \langle b,d \rangle \in S \}\ .$$

Recalling signature *Memory*, attribute map stands in Alloy for a ternary relation

$$\text{map} \subseteq Memory \times \text{addrs} \times \text{Data}\ .$$

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. We will in general denote the encoding of a relation C as a binary relation, by C. Given an object (in the relational sense — cf. 4.1.1) 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

$$a \bullet R = \breve{\pi}; Ran\,(a;R)\,;\rho\ . \tag{3}$$

Operation *Ran* in (3) returns the range of a relation as a partial identity. It is defined by

$$Ran\,(x) = (x;1) \cdot 1'\ .$$

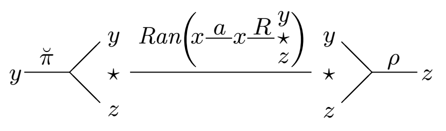
Its semantics in terms of binary relations is given by

$$Ran\,(R) = \{ \langle a,a \rangle : \exists b \text{ s.t. } \langle b,a \rangle \in R \}\ .$$

If we denote by $x \overset{R}{\text{---}} y$ the fact that x and y are related via the relation R, then Fig. 4 gives a graphical explanation of operation $\bullet$.

For a binary relation R representing a relation of rank 2, navigation is easier. Given a relational object $a : t$, we define

$$a \bullet R = Ran\,(a;R)\ .$$



y
$\breve{\pi}$
y
$\star$
z
$Ran\left(x \xrightarrow{a} x \xrightarrow{R} \begin{smallmatrix} y \\ \star \\ z \end{smallmatrix}\right)$
y
$\star$
z
ρ
z

Figure 4: Semantics of •

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

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

4.1.3 Translating Alloy Formulas to Fork Algebra Equations

In this section we will deal with the problem of translating a RL formula to a FRL equation in a way such that validity of the original formula in RL equivales proving the translation in the equational calculus FRL. Regarding Fig. 1, in this section we deal with the portion depicted in Fig. 5.

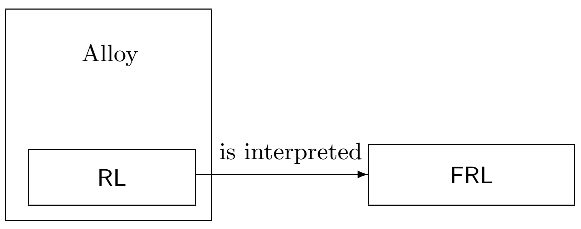


Figure 5: Relationships among the formalisms.

Atomic RL formulas can be seen as equations. If we can translate RL equations to FRL equations, we still have to deal with Boolean connectives and quantifiers. Fortunately, as we will show next, Boolean combinations of RL equations can be reduced to a single RL equation, and therefore can be easily translated to FRL. At the end of the section we will show how to handle quantified equations.

It is well known [39, p. 26] that Boolean combinations of relation algebraic equations can be translated into a single equation of the form $R = 1$. Since Alloy terms are typed, the translation must be modified slightly. We denote by 1 the untyped universal relation. By 1_k we will denote the universal k-ary relation. The transformation, for n-ary Alloy terms a and b, is:

$$a \text{ in } b \quad \rightsquigarrow \quad (1_n - a) + b = 1_n$$

For a formula of the form $!(a = 1_n)$, we reason as follows:

$$!(a = 1_n) \iff !(1_n - a = 0) \ .$$

Now, from a nonempty n-ary relation, we must generate a universal n-ary relation. Notice that if $1_n - a$ is nonempty, then $1_1.(1_n - a)$ is nonempty, and has arity $n-1$. Thus, the term

$$\underbrace{1_1.(\cdots .(1_1}_{n-1}.(1_n - a))\cdots)$$

yields a nonempty 1-ary relation. If we post compose it with 1_2, we obtain the universal 1-ary relation. If the resulting relation is then composed with the $(n+1)$-ary universal relation, we obtain the desired n-ary universal relation. We then have

$$!(a = 1_n) \quad \rightsquigarrow \quad (\underbrace{1_1.(\cdots .(1_1}_{n-1}.(1_n - a))\cdots).1_2).1_{n+1} = 1_n \ .$$

If we are given a formula of the form

$$a = 1_n \ \&\& \ b = 1_m,$$

with $n = m$, then the translation is trivial:

$$a = 1_n \ \&\& \ b = 1_m \quad \rightsquigarrow \quad a \& b = 1_n \ .$$

If $m > n$, we will convert a into a m-ary relation a' such that $a' = 1_m$ if and only if $a = 1_n$. Let a' be defined as

$$a.Id_3.1_{m-n+1} \ .$$

Then,

$$a = 1_n \ \&\& \ b = 1_m \quad \rightsquigarrow \quad a' \& b = 1_m \ .$$

Therefore, we will assume that whenever a quantifier occurs in a formula, it appears being applied to an equation of the form $t = 1_n$, where t is a RL term, and $n \in \mathbb{N}$. RL term t may contain variables. Since variables in RL stand for single objects, if term t contains variables $x_1, \ldots, x_m$, it will be translated to a term $T_m(t)$ such that

$$\begin{aligned} &\langle x, y\rangle \in t(b_1, \ldots, b_m) \\ \iff \quad &\langle (b_1 \star \cdots \star b_m) \star x, (b_1 \star \cdots \star b_m) \star y\rangle \in T_m(t) \ . \end{aligned}$$

If we define relations $X_i (1 \le i \le k)$ by

$$X_i = \begin{cases} \rho^{;(i-1)};\pi & \text{if } 1 \le i < k, \\ \rho^{;(i-1)} & \text{if } i = k \ , \end{cases}$$

an input $a_1 \star \cdots \star a_k$ is related through term X_i to a_i. Notice then that the term $Dom\,(\pi;X_i \ \cap \ \rho)$ filters those inputs $(a_1 \star \cdots \star a_k) \star b$ in which $a_i \neq b$ (i.e., the value b is bound to be a_i). The translation is defined as follows:

$$\begin{array}{lcl} T_m(C) & = & (Id_{S_1} \otimes \cdots \otimes Id_{S_m}) \otimes \mathsf{C}, \\ T_m(x_i) & = & Dom\,(\pi;X_i \ \cap \ \rho)\,, \\ T_m(r+s) & = & T_m(r) \cup T_m(s), \\ T_m(r\&s) & = & T_m(r) \cap T_m(s), \\ T_m(r-s) & = & T_m(r) \cap \overline{T_m(s)} \cap ((Id_{S_1} \otimes \cdots \otimes Id_{S_m}) \otimes 1)\,, \\ T_m(\sim r) & = & T_m(r)^{\smile}, \\ T_m(+r) & = & T_m(r);T_m(r)^{*}. \end{array}$$

In order to define the translation for navigation $r.s$ and application $s[v]$, we need to distinguish whether s is a binary relation, or if it has greater arity. The definition is as follows:

$$T_m(r.s) = \begin{cases} T_m(r) \bullet T_m(s) & \text{if } s \text{ is binary}, \\ T_m(r) \bullet (T_m(s); ((1\text{'} \otimes \pi) \nabla (1\text{'} \otimes \rho))) & \text{otherwise.} \end{cases}$$

$$T_m(s[v]) = \begin{cases} T_m(v) \bullet T_m(s) & \text{if } s \text{ is binary}, \\ T_m(v) \bullet (T_m(s); ((1\text{'} \otimes \pi) \nabla (1\text{'} \otimes \rho))) & \text{otherwise.} \end{cases}$$

In case there are no quantified variables, there is no need to carry the values on, and the translation becomes:

$$\begin{array}{lcl}
T_0(C) & = & \mathsf{C}, \\
T_0(r{+}s) & = & T_0(r) \cup T_0(s), \\
T_0(r\&s) & = & T_0(r) \cap T_0(s), \\
T_0(r-s) & = & T_0(r) \cap \overline{T_0(s)}, \\
T_0(\sim r) & = & T_0(r)\breve{}, \\
T_0(+r) & = & T_0(r);T_0(r)^*, \\
T_0(r.s) & = & T_0(r) \bullet T_0(s), \\
T_0(s[r]) & = & T_0(r) \bullet T_0(s)\ .
\end{array}$$

It is now easy to prove a theorem establishing the relationship between RL terms and their corresponding translation. Notice that for every environment e:

- Given a type T, $e(T)$ is a nonempty set.
- Given a variable v, $e(v)$ is a n-ary relation for some $n \in \mathbb{N}$.

We define the environment e' 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,

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

For the 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 [24]. 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, use of navigation in object-oriented settings usually falls in the situation modeled by us. With the aim of using a shorter notation, the value (according to the standard semantics) of a term t in an environment e will be denoted by $e(t)$ rather than by $X[t]e$. Similarly, the value in FRL of a term t in an environment e' will be denoted by $e'(t)$ rather than by $Y[t]e'$. In order to simplify notation, we will denote by $b^\star$ the element $b_1 \star \cdots \star b_m$.

THEOREM 4.1. *For every Alloy term t such that:*

1. *$X[t]e$ defines a n-ary relation,*
2. *there are m free variables $x_1, \ldots, x_m$ in t,*

$$Y[T_m(t)]e' = \begin{cases} \{\langle b^\star \star a, b^\star \star a\rangle : \\ \qquad a \in X[t]e(\overline{b} \mapsto \overline{x})\} & \textit{if } n = 1 \\ \{\langle b^\star \star a_1, b^\star \star (a_2 \star \cdots \star a_n)\rangle : \\ \qquad \langle a_1, \ldots, a_n\rangle \in X[t]e(\overline{b} \mapsto \overline{x})\} & \textit{if } n > 1 \end{cases}$$

PROOF. The proof follows by induction on the structure of term t. As a sample we prove it for the variables, the remaining cases being simple applications of the semantics of the fork algebra operators.

If v is a quantified variable (namely, x_i), then $e(t)$ is a unary relation.

$$\begin{array}{ll}
e'\left(T_m(x_i)\right) & \\
\quad = e'\left(Dom\,(\pi;X_i\ \cap\ \rho)\right) & \text{(by def. } T_m) \\
\quad = \{\, \langle b^\star \star a, b^\star \star a\rangle : a = b_i\,\} & \text{(by semantics)} \\
\quad = \{\, \langle b^\star \star a, b^\star \star a\rangle : a \in \{\, b_i\,\}\,\} & \text{(by set theory)} \\
\quad = \left\{\, \langle b^\star \star a, b^\star \star a\rangle : a \in \left(e(\overline{b} \mapsto \overline{x})\right)(x_i)\,\right\}\ . & \\
 & \text{(by def. } e(\overline{b} \mapsto \overline{x}))
\end{array}$$

If v is a variable distinct of $x_1, \ldots, x_m$, there are two possibilities.

1. $e(v)$ denotes a unary relation.
2. $e(v)$ denotes a n-ary relation with $n > 1$.

If $e(v)$ denotes a unary relation,

$$\begin{array}{ll}
e'\left(T_m(v)\right) & \\
\quad = e'\left((Id_{S_1} \otimes \cdots \otimes Id_{S_m}) \otimes v\right) & \text{(by def. } T_m) \\
\quad = (Id_{S_1} \otimes \cdots \otimes Id_{S_m}) \otimes e'(v) & \text{(by semantics)} \\
\quad = (Id_{S_1} \otimes \cdots \otimes Id_{S_m}) \otimes \{\, \langle a,a\rangle : a \in e(v)\,\} & \text{(by def. } e') \\
\quad = \{\, \langle b^\star \star a, b^\star \star a\rangle : a \in e(v)\,\} & \text{(by semantics)} \\
\quad = \left\{\, \langle b^\star \star a, b^\star \star a\rangle : a \in (e(\overline{b} \mapsto \overline{x})))(v)\,\right\}\ . & \\
 & \text{(by def. } e(\overline{b} \mapsto \overline{x}))
\end{array}$$

If $e(v)$ denotes a n-ary relation $(n > 1)$,

$$\begin{array}{ll}
e'\left(T_m(v)\right) & \\
\quad = e'\left((Id_{S_1} \otimes \cdots \otimes Id_{S_m}) \otimes v\right) & \text{(by def. } T_m) \\
\quad = (Id_{S_1} \otimes \cdots \otimes Id_{S_m}) \otimes e'(v) & \text{(by semantics)} \\
\quad = (Id_{S_1} \otimes \cdots \otimes Id_{S_m}) \otimes & \\
\qquad \{\, \langle a_1, a_2 \star \cdots \star a_n\rangle : \langle a_1, \ldots, a_n\rangle \in e(v)\,\} & \text{(by def. } e') \\
\quad = \{\, \langle b^\star \star a_1, b^\star \star (a_2 \star \cdots \star a_n)\rangle : \langle a_1, \ldots, a_n\rangle \in e(v)\,\} & \\
 & \text{(by semantics)} \\
\quad = \{\langle b^\star \star a_1, b^\star \star (a_2 \star \cdots \star a_n)\rangle\ : & \\
\qquad \langle a_1, \ldots, a_n\rangle \in (e(\overline{b} \mapsto \overline{x})))(v)\}\ . & \text{(by def. } e(\overline{b} \mapsto \overline{x}))
\end{array}$$

■

In order to translate a RL formula α we will assume the following:

- If a subformula of α is a Boolean combination of atomic formulas, then, before translating α, β has been converted to a single equation of the form $R = 1_n$ following the procedure explained at the beginning of this section.
- Before translating α, all the negations have been pushed into the formula as much as possible, using simple valid transformations such as:

$$\begin{array}{c}
\neg\neg\beta \rightsquigarrow \beta, \\
\neg(\beta \vee \gamma) \rightsquigarrow \neg\beta\ \wedge\ \neg\gamma, \\
\neg(\beta \wedge \gamma) \rightsquigarrow \neg\beta\ \vee\ \neg\gamma, \\
\neg(\exists x : S)\beta \rightsquigarrow (\forall x : S)\neg\beta, \\
\neg(\forall x : S)\beta \rightsquigarrow (\exists x : S)\neg\beta,
\end{array}$$

Notice that this implies that negations will only appear next to atomic formulas. Therefore, in virtue of the item above, no negation appears in α at all.

In the next paragraphs we will define a mapping T'_m (where m is the number of variables that might occur free in the formula being translated) that will allow us to translate RL formulas to FRL terms. We then define function $\mathsf{RL} \mapsto \mathsf{FRL}$ (mapping RL sentences to FRL equations) on a sentence α by the condition

$$\mathsf{RL} \mapsto \mathsf{FRL}(\alpha) \stackrel{def}{=} T'_0(\alpha) = 1 \ .$$

atomic: Let α be the atomic formula $t = 1_n$ (where m variables $x_1, \ldots, x_m$ occur free in term t). Notice that according to Thm. 4.1, $T_m(t)$ is a binary relation whose elements are pairs

$$\langle (b_1 \star \cdots \star b_m) \star a_1, (b_1 \star \cdots \star b_m) \star (a_2 \star \cdots \star a_n) \rangle \ .$$

From the set-theoretical definition of fork and the remaining relational operators, it follows that[5]

$$\begin{aligned}
&\langle (b_1 \star \cdots \star b_m) \star a_1, (b_1 \star \cdots \star b_m) \star (a_2 \star \cdots \star a_n) \rangle \in T_m(t) \\
&\iff \\
&((b_1 \star \cdots \star b_m) \star a_1) \star (a_2 \star \cdots \star a_n) \in \mathsf{ran}\,(Id \ \nabla \ T_m(t) ; \rho) \\
&\iff \\
&\langle ((b_1 \star \cdots \star b_m) \star a_1) \star (a_2 \star \cdots \star a_n), c \rangle \in \\
&\qquad\qquad\qquad Ran\,(Id \ \nabla \ T_m(t) ; \rho)\,; 1 \ .
\end{aligned}$$

Formula α states that every n-tuple belongs to the semantics of t. Therefore, we must quantify universally over all values $a_1, a_2, \ldots, a_n$. We define (for a variable x_i of sort S) the relational term $\exists_{x_i}$ as follows[6]:

$$\exists_{x_i} = X_1 \nabla \cdots \nabla X_{i-1} \nabla 1_S \nabla X_{i+1} \nabla \cdots \nabla X_k \ .$$

For instance, if $k = 3$, we have $\exists_{x_2} = X_1 \nabla 1_S \nabla X_3$. This term defines the binary relation

$$\{ \langle a_1 \star a_3, a_1 \star a_2 \star a_3 \rangle : a_2 \in S \} \ .$$

Notice that term $\exists_{x_2}$ generates all possible values for variable x_2. Given a term t standing for a binary relation with one free variable, the term

$$\exists_{x_2} ; Ran\,(Id \ \nabla \ T_1(t) ; \rho)\,; 1$$

describes the binary relation

$$\{ \langle b_1 \star a_2, c \rangle : (\exists a_1 : S)\,(\langle b_1 \star a_1, b_1 \star a_2 \rangle \in T_1(t)) \} \ .$$

We define $\dagger(t) := Ran\,(Id \ \nabla \ T_m(t) ; \rho)\,; 1$. Term $\exists_{x_2} ; \dagger(t)$ indeed quantifies variable x_2 existentially over the domain S. Profiting from the interdefinability of $\exists$ and $\forall$, the term

$$(Id \otimes Id) ; \overline{\exists_{x_2} ; \overline{\dagger(t)}}$$

allows us to quantify variable x_2 universally. We will denote such a term as $\forall_{x_2} \dagger(t)$.

We then define $T'_m(t = 1_n)$ as $(\forall a_1) \cdots (\forall a_n) \dagger(t)$.

conjunction: Let $\alpha = \beta \&\& \gamma$. Let m be the maximum number of variables over individuals free in either β or γ. Let $t_1 := T'_m(\beta)$, $t_2 := T'_m(\gamma)$. We then define $T'_m(\alpha) = t_1 \cap t_2$.

[5]Given a binary relation R, $\mathsf{ran}\,(R)$ denotes the set $\{ b : \exists a \text{ such that } \langle a, b \rangle \in R \}$.
[6]We define relation 1_S as $1 ; Id_S$, the universal binary relation whose range is restricted to sort S.

disjunction: Let $\alpha = \beta \ || \ \gamma$. Let m be the maximum number of variables over individuals free in either β or γ. Let $t_1 := T'_m(\beta)$, $t_2 := T'_m(\gamma)$. We then define $T'_m(\alpha) = t_1 \cup t_2$.

existential: Let $\alpha = \text{some } x_i : S \mid \beta$. We define $T'_m(\alpha) = \exists_{x_i} ; T'_{m+1}(\beta)$. Moving from T'_m to T'_{m+1} is justified because there may be a new free variable in β, namely, x_i.

universal: Let $\alpha = \text{all } x_i : S \mid \beta$. We define $T'_m(\alpha) = \forall_{x_i} T'_{m+1}(\beta)$. Moving from T'_m to T'_{m+1} is justified because there may be a new free variable in β, namely, x_i.

Once the translation T'_m has been defined, the following theorem, showing the adequacy of the translation, can be proved by induction on the structure of RL formulas.

THEOREM 4.2. *For every RL sentence α, for every environment e,*

$$M[\alpha]e \quad \iff \quad N[\mathsf{RL} \mapsto \mathsf{FRL}(\alpha)]e' ,$$

where environment e' is defined as in Thm. 4.1.

Example: Let us consider the following assertion:

$$\text{some s} : \mathit{System} \mid \text{s.cache.map in s.main.map} \ . \qquad (4)$$

Once converted to an equation of the form $R = 1$, assertion (4) becomes

$$\begin{aligned}
&\text{some s} : \mathit{System} \mid \\
&\qquad (1_2 - \text{s.cache.map}) + \text{s.main.map} = 1_2 \ . \qquad (5)
\end{aligned}$$

If we apply translation T_1 to the term on the left-hand side of the equality in (5), it becomes

$$\begin{aligned}
&\begin{pmatrix} Id_S \\ \otimes \\ 1 \end{pmatrix} \cap \overline{Dom\,(\pi ; X_{\text{s}} \cap \rho) \bullet \begin{matrix} Id_S \\ \otimes \\ \text{cache} \end{matrix} \bullet \begin{pmatrix} Id_S & & Id \otimes \pi \\ \otimes & ; & \nabla \\ \text{map} & & Id \otimes \rho \end{pmatrix}} \\
&\qquad \cup \ Dom\,(\pi ; X_{\text{s}} \cap \rho) \bullet \begin{matrix} Id_S \\ \otimes \\ \text{main} \end{matrix} \bullet \begin{pmatrix} Id_S & & Id \otimes \pi \\ \otimes & ; & \nabla \\ \text{map} & & Id \otimes \rho \end{pmatrix} \ . \qquad (6)
\end{aligned}$$

Since s is the only variable, $X_{\text{s}} = \rho^0 = Id$, and therefore (6) becomes

$$\begin{aligned}
&\begin{pmatrix} Id_S \\ \otimes \\ 1 \end{pmatrix} \cap \overline{Dom\,(\pi \cap \rho) \bullet \begin{matrix} Id_S \\ \otimes \\ \text{cache} \end{matrix} \bullet \begin{pmatrix} Id_S & & Id \otimes \pi \\ \otimes & ; & \nabla \\ \text{map} & & Id \otimes \rho \end{pmatrix}} \\
&\qquad \cup \ Dom\,(\pi \cap \rho) \bullet \begin{matrix} Id_S \\ \otimes \\ \text{main} \end{matrix} \bullet \begin{pmatrix} Id_S & & Id \otimes \pi \\ \otimes & ; & \nabla \\ \text{map} & & Id \otimes \rho \end{pmatrix} \ . \qquad (7)
\end{aligned}$$

Certainly (7) is harder to read than the equation in (4). This can probably be improved by adding appropriate syntactic sugar to the language. Let us denote by E the term in (7). Following our recipe, applying $\mathsf{RL} \mapsto \mathsf{FRL}$ we arrive to the following equation

$$\exists_{\text{s}} ; \forall_x \forall_y \,(Ran\,(Id \ \nabla \ E ; \rho)\,; 1) = 1,$$

which can now be verified equationally in FRL.

4.1.4 Analyzing FRL

An essential feature of Alloy is its adequacy for automatic analysis. It is clear that the translation defined in Section *4.1.3* induces a new semantics for RL formulas in terms of fork algebras. That is, given a RL formula α whose FRL translation is a fork term t_α, we can compute the semantics of t_α using function N (cf. Fig. 3). Thus, an immediate question is what is the impact of this new semantics in the analysis of Alloy specifications.

In the next paragraphs we will argue that the new semantics can fully profit from the analysis procedure provided by the Alloy Analyzer. Notice that the Alloy Analyzer 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.

Given a specification whose types are $T_1, \ldots, T_n$, and a counterexample assigning to each type T_i a domain D_i, let D be defined as $\bigcup_{1 \leq i \leq n} D_i$.

Once D is defined, we define the set $D^\star$ by $\bigcup_{0 \leq i} D_i^\star$, where

$$\begin{array}{rcl} D_0^\star & = & D \ , \\ D_{n+1}^\star & = & D_n^\star \cup \{ a \star b : a, b \in D_n^\star \} \ . \end{array}$$

Let us consider now the proper fork algebra $\mathfrak{A}$ whose domain is $\mathcal{P}(D^\star \times D^\star)$, and whose forking operation is defined, for binary relations R and S, by

$$R \nabla S = \{ \langle a, b \star c \rangle : a \; R \; b \ \wedge \ a \; S \; c \} \ .$$

Given a counterexample (environment) e according to the standard semantics of Alloy, we will build a counterexample e' according to the fork-algebraic semantics. Notice that for an Alloy sentence α and an environment e, Thm. 4.2 shows that

$$M[\alpha]e \quad \Longleftrightarrow \quad N[\mathsf{RL} \mapsto \mathsf{FRL}(\alpha)]e',$$

where environment e' is defined as in Thm. 4.1. Environment e' is the sought counterexample.

This shows that all the work that has been done so far toward the analysis of Alloy specifications can be used toward the verification of Alloy specifications with respect to the new 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 in Thm. 4.2 we assume that that the fork algebra on which the environment assigns values to variables, is proper. So the question arises of whether using non proper fork algebras allows one to verify the same properties that the Alloy Analyzer does. Actually, the surprising answer is that it is possible to verify *strictly more* properties. That is, there exists at least one problem for which the Alloy Analyzer cannot find a counterexample (no matter the scopes chosen), but for which a small counterexample exists using non proper fork algebras. In the next paragraphs we will discuss this briefly. A complete discussion exceeds the scope of this paper.

The Specification: Assume as given an Alloy specification stating that a binary relation R is a total ordering.
The Assertion: There are first and last elements for the ordering R.

This assertion is flawed. Any infinite total ordering provides a counterexample. Unfortunately, since the Alloy Analyzer can only handle finite relations, and every finite total ordering is bounded, no counterexample will be found, no matter the scope chosen.

On the other hand, there is a finite representable relation algebra (actually, it has 8 elements), in which there is a relation that in every representation is a dense linear order without end points. This relation provides the counterexample.

4.1.5 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. We begin with an example. Recalling the specification of function Flush in Section 2.1, the specification has the shape

$$\text{some x : set t / F .} \tag{8}$$

This is recognized within Alloy as a higher-order formula [20]. Let us analyze what happens in the modified semantics. Since t is a type (set), then it stands for a subset of Id. Similarly, subsets of t are subsets of the identity, which are contained in t. Thus, formula (8) 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

$$\textit{some } s : \textit{state}/\textit{op}1(\textit{pre}, s) \textit{ and } \textit{op}2(s, \textit{post}) \tag{9}$$

is first-order with the modified semantics. Operations *op*1 and *op*2 can be defined as binary predicates in a first-order language for fork algebras, and thus formula (9) is first-order.

So far this result only shows that the newly defined semantics fits better with the language than the standard one. We are currently working on the application of the new semantics in verification.

5. ADDING DYNAMIC FEATURES TO ALLOY

In this section we extend Alloy's relational logic syntax and semantics with the aim of dealing with properties of executions of operations specified in Alloy. Recalling Fig. 1, in this section we will deal with the portion reproduced in Fig. 6.

The reason for this extension (called DynAlloy) is that we want to provide a setting in which, besides functions describing sets of states, there are actions that actually change

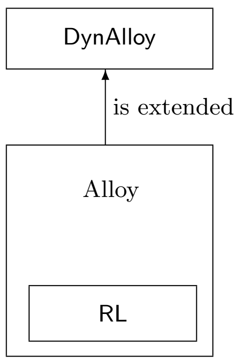


Figure 6: Alloy and its dynamic extension DynAlloy.

states (i.e., they describe relations between input and output data). Actions are built from atomic actions using well known constructs for sequential programming languages. We will describe the syntax and semantics of DynAlloy in Section 5.1, but it is worth mentioning at this point that both were strongly motivated by dynamic logic [18], and the suitability of dynamic logic for expressing partial and total correctness assertions. In Section 5.2 we propose a proof method for dealing with properties of executions. In Section 5.3 we show how to analyze properties of executions using the Alloy analyzer, by first computing the weakest liberal precondition of actions [10]. Finally, in Section 5.4 we present a short case-study as an example of how this method can be applied to prove properties of executions of Alloy specifications.

5.1 Functions vs. Actions

Functions in Alloy are just parameterized formulas. Some of the parameters are considered input parameters, and the relationship between input and output parameters is carried out by the convention that the second argument is the result of the function application. Following [24], the function *dom* that yields the domain of a relation is defined as

$$\mathsf{sig}\ X\ \{\} \quad \mathsf{fun}\ dom\ (r : X \to X, d : X)\{d = r.X\}\ . \tag{10}$$

Then, if α is a formula with one free variable and we want to prove that α holds when applied to the domain of the relation r, (10) is used as follows:

$$\mathsf{all}\ result : X \mid dom(r, result) \Rightarrow \alpha(result)\ .$$

Notice that there is no real change in the state of the system, since no variable actually changes its value.

Dynamic logic [18], arose in the early '70s with the intention of faithfully reflecting state change. In the following paragraphs we propose, motivated by its syntax, the use of actions to model state change in Alloy.

What we would like to say about an action is how it transforms the system state after its execution. We can do this by using pre and post conditions. An assertion of the form

$$\begin{array}{c}\alpha\\ \{A\}\\ \beta\end{array}$$

affirms that whenever action A is executed on a state satisfying α, if it terminates, it does so in a state satisfying β. This approach is particularly appropriate, since behaviors described by functions are better viewed as the result of performing an action on an input state. Thus, a definition of the function *dom* has as counterpart a definition of an action *DOM* of the form

$$\begin{array}{c}r = r_0 \wedge d = d_0\\ \{DOM(r, d)\}\\ r = r_0 \wedge d = r.X\ .\end{array} \tag{11}$$

Although it may be hard to find out what are the differences between (10) and (11) just by looking at the formulas (i.e., both formulas seem to provide the same information), the differences rely in the semantics, as well as the fact that actions can be sequentially composed, iterated or nondeterministically chosen, while Alloy functions cannot.

The syntax of Alloy's formulas is the same presented in Fig. 2, with the addition of the following clause for building partial correctness statements (we assume that pre and post conditions are RL formulas):

$$\begin{array}{lll} formula & ::= & \ldots \mid formula\ \{program\}\ formula\\ & & \quad\text{"partial correctness"}\end{array}$$

The syntax for programs is the one defined in [18] for the class of regular programs plus a new rule to allow the construction of atomic actions from their pre and post conditions.

$$\begin{array}{lllr} program & ::= & \langle formula, formula\rangle & \text{"atomic action"}\\ & \mid & formula? & \text{"test"}\\ & \mid & program + program & \text{"non-deterministic choice"}\\ & \mid & program ; program & \text{"sequential composition"}\\ & \mid & program^{*} & \text{"iteration"}\end{array}$$

In Fig. 7 we extend the definition of function M to partial correctness assertions and define the denotational semantics of programs as binary relations over *env*. The definition of function M on a partial correctness assertion makes clear that we are actually choosing partial correctness semantics. This follows from the fact we are not requesting environment e to belong to the domain of the relation $P[p]$. In order to assign semantics to atomic actions, we will assume there is a function A assigning to each atomic action a binary relation on the environments. We impose the following restriction on A:

$$A(\langle pre, post\rangle) \subseteq \{\ \langle e, e'\rangle : M[pre]e \wedge M[post]e'\ \}\ .$$

There is a subtle point in the definition of the semantics of atomic programs. We assume that actions modify certain variables, and those variables that are not modified retain their values. Thus, given an atomic action

$$\begin{array}{c}x = x_0\\ \{Add1\}\\ x = x_0 + 1\end{array}$$

adding 1 to the value of parameter x, it is clear that variable x_0 must retain its value. Without this assumption, the definition we provide accepts awkward pairs of environments $\langle e, e'\rangle$ satisfying, for instance, $e(x) = e(x_0) = 0$, and $e'(x) = 11$ and $e'(x_0) = 10$.

5.2 Specifying and Proving Properties of Executions: Motivation

Suppose we want to show that a given property P is invariant under sequences of applications of the operations

$M[\alpha\{p\}\beta]e = M[\alpha]e \implies \forall e' (\langle e, e' \rangle \in P[p] \implies M[\beta]e')$

$P : program \to \mathcal{P}(env \times env)$

$P[\langle pre, post \rangle] = A(\langle pre, post \rangle)$
$P[\alpha?] = \{ \langle e, e' \rangle : M[\alpha]e \wedge e = e' \}$
$P[p_1 + p_2] = P[p_1] \cup P[p_2]$
$P[p_1 ; p_2] = P[p_1] ; P[p_2]$
$P[p^*] = P[p]^*$

Figure 7: Syntax and Semantics of DynAlloy.

"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 $O \in \{Flush, SysWrite\}$ that

$$P(s) \wedge O(s, s') \Rightarrow P(s') \ .$$

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 the introduction of *traces* in Alloy [24].

The following example, extracted from [24], shows signatures for clock ticks and for traces of states. The first exclamation mark in the definition of next means it is total on its declared domain.

```
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 modeling perspective it is less than 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.

The specification in DynAlloy of actions *SysWrite* and *Flush* is done as follows:

```
s = s_0

   {SysWrite(s: System)}

some d: Data, a: Addr |
   s.cache = s_0.cache ++ (a → d) ∧
   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 partial correctness statements on the set of regular programs generated by the set of atomic actions $\{ SysWrite, Flush \}$, we can assert the invariance of a property P under finite applications of functions SysWrite and Flush as follows:

$$\begin{gathered} Init(s) \wedge P(s) \\ \{ \ (SysWrite(s) + Flush(s))^* \ \} \\ P(s) \ . \end{gathered}$$

More generally, suppose now that we want to show that a 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

$$\begin{gathered} Init \wedge Q \\ \{ \ (O_1 + \cdots + O_k)^* \ \} \\ Q \ . \end{gathered} \tag{12}$$

Notice that there is no need to mention traces in the specification of the previous properties. This is because finite traces get determined by the semantics of reflexive-transitive closure.

5.3 Analysis of DynAlloy Specifications

As we mentioned throughout the paper, Alloy's design was deeply influenced by the intention of producing an automatically analyzable language. While DynAlloy is better suited than Alloy for the specification of properties of executions, the use of ticks and traces allows one to automatically analyze properties of executions. Therefore, an almost mandatory question is whether DynAlloy can be automatically analyzed, and if so, what is the effort required to this end. In this section we show how to analyze DynAlloy automatically using the Alloy analyzer. In [19], a function

$$MT : formula \to booleanformulatree$$

allows one to transform Alloy formulas to Boolean formulas. These formulas are later on transformed into conjunctive normal form and fed to off-the-shelf SAT-solvers. The main rationale behind our technique is the translation of partial correctness assertions to first-order Alloy formulas, using weakest liberal preconditions [10].

In the next paragraphs we will define a function

$$wlp : program \times formula \rightarrow formula$$

that computes the weakest liberal precondition of a formula according to a program. We will in general use names $x_1, x_2 \ldots$ for program variables, and will use names $y_1, y_2, \ldots$ for rigid variables, i.e., those auxiliary variables whose values are not affected by actions. We will denote by $\alpha|_x^v$ the substitution of variable x by the fresh variable v in formula α. For an atomic action $\langle pre, post\rangle$ we assume $\overline{y} = y_1, \ldots, y_k$ are the rigid variables, and $\overline{x} = x_1, \ldots, x_n$ the program variables. Function *wlp* is then defined as follows:

$$\begin{array}{lcl}
wlp[\langle pre, post\rangle\ , f] & = & \text{all}\ \overline{y}\left(pre \implies \text{all}\ \overline{v}\left(post|_{\overline{x}}^{\overline{v}} \implies f|_{\overline{x}}^{\overline{v}}\right)\right) \\
wlp[g?, f] & = & g \implies f \\
wlp[p_1 + p_2, f] & = & wlp[p_1, f] \wedge wlp[p_2, f] \\
wlp[p_1 ; p_2, f] & = & wlp[p_1, wlp[p_2, f]] \\
wlp[p^*, f] & = & \bigwedge_{i=0}^{\infty} wlp[p^i, f]\ .
\end{array}$$

Notice that *wlp* yields Alloy formulas in all cases except for the iteration construct, where the resulting formula may be infinitary. In order to obtain an Alloy formula, we can impose a bound on the depth of iterations. This equivales to fixing a maximum length for traces. A function *Bwlp* (bounded weakest liberal precondition) is then defined as *wlp* except for iteration, where it is defined by:

$$Bwlp[p^*, f] = \bigwedge_{i=0}^{n} Bwlp[p^i, f]\ . \tag{13}$$

In (13), n is the scope set for the depth of iteration.

We now extend the definition of function MT to partial correctness statements by the condition:

$$MT[\alpha\ \{p\}\ \beta] = MT[\alpha \implies Bwlp[p, \beta]]\ .$$

Of course this proof method is not complete, but clearly it is not meant to be; from the very beginning we placed restrictions on the domains involved in the specification to be able to turn first-order formulas into propositional formulas. This is just another step in the same direction.

5.4 A short case-study

In this section we will develop a short case-study to show how this proof method is used. As an instance of (12), 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. This property is specified in DynAlloy by:

$$\begin{array}{l}
\text{all}\ s : \text{System}\ | \\
\qquad\qquad \text{DirtyInv}(s) \\
\qquad \{(\text{SysWrite}(s) + \text{Flush}(s))^*\} \\
\qquad\qquad \text{DirtyInv}(s)\ .
\end{array} \tag{14}$$

Notice also that if after finitely many executions of SysWrite and Flush we flush <u>all</u> the dirty addresses in the cache to main memory, the resulting cache should fully agree with main memory. In order to specify this property we need to specify the function that flushes all the dirty cache addresses. The specification is as follows:

$$\begin{array}{l}
\text{s} = \text{s}_0 \\
\\
\quad \{\text{DSFlush}(\text{s} : \textit{System})\} \\
\\
\text{s.cache.dirty} = \emptyset\ \wedge \\
\quad \text{s.cache.map} = \text{s}_0\text{.cache.map} - \\
\qquad \text{s}_0\text{.cache.map}[\text{s}_0\text{.cache.dirty}]\ \wedge \\
\quad \text{s.main.map} = \text{s}_0\text{.main.map} ++ \\
\qquad \text{s}_0\text{.cache.map}[\text{s}_0\text{.cache.dirty}]
\end{array}$$

We specify the property establishing the agreement of the cache with main memory as follows:

$$\begin{array}{l}
\text{FullyAgree}(s : \textit{System}) \\
\qquad\qquad \iff s\text{.cache.map}\ \textit{in}\ s\text{.main.map}\ .
\end{array}$$

Once "DSFlush" and "FullyAgree" have been specified, the property is specified in DynAlloy by:

$$\begin{array}{l}
\text{all}\ s : \text{System}\ | \\
\qquad\qquad \text{DirtyInv}(s) \\
\qquad \{(\text{SysWrite}(s) + \text{Flush}(s))^* ; \text{DSFlush}(s)\} \\
\qquad\qquad \text{FullyAgree}(s)\ .
\end{array} \tag{15}$$

Now, it only remains to apply function MT to formula (15) and feed the Alloy analyzer with the resulting formula.

6. A COMPLETE CALCULUS FOR DYNAMIC ALLOY

In this section we present a complete calculus for reasoning about properties specified in DynAlloy. Recalling Fig. 1, in this Section we deal with the portion reproduced in Fig. 8.

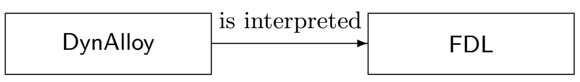


Figure 8: Relationships among the formalisms DynAlloy and FDL.

The formalism FDL (Fork Dynamic Logic) can be succinctly described as first-order dynamic logic over the equational theory of fork algebras. The reason for using dynamic logic is that there is a close relationship between this logic and partial correctness assertions. In Section 6.1 we present the syntax and semantics of first-order dynamic logic. In Section 6.2, dynamic logic is extended with fork algebras. We then extend function RL $\mapsto$ FRL so that it also translates partial correctness assertions. Finally, in Section 6.3 we present a complete calculus for FDL.

6.1 Dynamic Logic

Dynamic logic is a formalism for reasoning about programs. From a set of atomic actions (usually assignments of terms to variables), and using appropriate 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 is 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 section 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. 9.

action ::= $a_1, \ldots a_k$ (atomic actions)
| *skip*
| action+action (nondeterministic choice)
| action;action (sequential composition)
| action* (finite iteration)
| dform? (test)

expr ::= *var*
| $f(\text{expr}_1, \ldots, \text{expr}_k)$ ($f \in F$ with arity k)

dform ::= $p(\text{expr}_1, \ldots, \text{expr}_n)$ ($p \in P$ with arity n)
| !dform (negation)
| dform && dform (conjunction)
| dform || dform (disjunction)
| *all* v : *type*/dform (universal)
| *some* v : *type*/dform (existential)
| [action]dform (box)

Figure 9: Syntax of dynamic logic

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. 10.

Q : form $\rightarrow ST \rightarrow Boolean$
P : action $\rightarrow \mathcal{P}(ST \times ST)$
Z : expr $\rightarrow ST \rightarrow \mathbf{s}$

$Q[p(t_1, \ldots, t_n)]\mu = (Z[t_1]\mu, \ldots, Z[t_n]\mu) \in env(p)$ (atomic formula)
$Q[!F]\mu = \neg Q[F]\mu$
$Q[F \&\& G]\mu = Q[F]\mu \wedge Q[G]\mu$
$Q[F \mid\mid G]\mu = Q[F]\mu \vee Q[G]\mu$
$Q[all\ v : t\ /\ F]\mu = \bigwedge \{Q[F](\mu \oplus v \mapsto x)/x \in env(t)\}$
$Q[some\ v : t\ /\ F]\mu = \bigvee \{Q[F](\mu \oplus v \mapsto x)/x \in env(t)\}$
$Q[\ [a]F\ ]\mu = \bigwedge \{Q[F]\nu / \langle \mu, \nu \rangle \in P(a)\}$

$P[a] = env(a)$ (atomic action)
$P[skip] = \{ \langle \mu, \mu \rangle : \mu \in ST \}$
$P[a + b] = P[a] \cup P[b]$
$P[a\,;b] = P[a] \circ P[b]$
$P[a^*] = (P[a])^*$
$P[\alpha?] = \{ \langle \mu, \mu \rangle : Q[\alpha]\mu \}$

$Z[v]\mu = \mu(v)$
$Z[f(t_1, \ldots, t_k)]\mu = env(f)(Z[t_1]\mu, \ldots, Z[t_k]\mu)$

Figure 10: Semantics of dynamic logic

6.2 FDL: Dynamic Logic over Fork Algebras

In order to define FDL, we include in the set of function symbols of signature Σ the set of constants $\{ 0, 1, 1\text{'} \}$, the set of unary symbols $\{ ^-, ^\smile \}$, and the set of binary symbols $\{ +, \cdot, ;, \nabla \}$. The only predicate we will consider is equality. Since these signatures include all operation symbols from fork algebras, they will be called *fork signatures*.

Remark 1. *Notice that FDL atomic formulas are equalities between fork algebra terms, and thus, for atomic formulas, function Q from Fig. 10 and function N from Fig. 3 agree.*

We will call theories containing the identities specifying the class of fork algebras *FDL theories*. By working with FDL theories we intend to describe structures for dynamic logic whose domains are sets of binary relations. This is indeed the case as it is shown in the following theorem.

THEOREM 6.1. *Let Σ be a fork signature, and Ψ be a FDL theory. For each model $\mathfrak{A}$ for Ψ there exists a model $\mathfrak{B}$ for Ψ, isomorphic to $\mathfrak{A}$, in which the domain $\mathbf{s}$ is a set of binary relations.*

PROOF. Let us consider the reduct $\mathcal{A}$ of the model $\mathfrak{A}$, obtained by keeping $\mathfrak{A}$'s domain and the fork algebra operations. $\mathcal{A}$ is a structure of the form $\langle A, +, \cdot, ^-, 0, 1, ;, ^\smile, 1\text{'}, \nabla \rangle$ in which the semantics of action, function and predicate symbols is given through an environment *env*. Since Ψ is a FDL theory (and therefore satisfies the axioms for fork algebras), $\mathcal{A}$ is a fork algebra. Thus, by [13][14, Thm. 4.2], $\mathfrak{A}$ is isomorphic to a proper fork algebra with domain B. Let $h : A \rightarrow B$ be the isomorphism. In order to define the model $\mathfrak{B}$ we will define an environment env' for action, function and predicate symbols as follows:

– Actions: $\langle s, s' \rangle \in env'(a) \iff \langle h^{-1}(s), h^{-1}(s') \rangle \in env(a)$, where for a state s, $h^{-1}(s)$ is the state satisfying $(h^{-1}(s))(v) = h^{-1}(s(v))$.

– Functions: $[env'(f)]\,(b) = h\left([env(f)]\,(h^{-1}(b))\right)$.

– Predicates: $b \in env'(p) \iff h^{-1}(b) \in env(p)$.

By construction, $\mathfrak{B}$ is isomorphic to $\mathfrak{A}$. ■

The previous theorem is essential, and its proof (which uses [14, Thm. 4.2]), heavily relies on the use of fork algebras rather than plain relation algebras [39]. A model for a FDL theory Ψ is a structure satisfying all the formulas in Ψ. Such a structure can, or cannot, have binary relations in its domain. Theorem 6.1 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 FDL, and interpret them as properties predicating about binary relations.

We will end this section by presenting the extension of function RL $\mapsto$ FRL to partial correctness assertions. The extension, which is defined as RL $\mapsto$ FRL for the remaining formula patterns, is denoted by DynAlloy $\mapsto$ FDL. Then,

$$\text{DynAlloy} \mapsto \text{FDL}\,(\alpha\{p\}\beta) = \\ \text{RL} \mapsto \text{FRL}(\alpha) \implies [p]\ \text{RL} \mapsto \text{FRL}(\beta)\ .$$

In the following paragraphs we present a theorem describing the relationship established by the translation, between the formalisms DynAlloy and FDL.

LEMMA 6.2. *Let α be a DynAlloy formula. Let e be a DynAlloy environment, and A the function that assigns meaning to atomic actions. Then, there exists a FDL environment $\widehat{e}$ such that*

$$M[\alpha]e = Q[\mathsf{DynAlloy} \mapsto \mathsf{FDL}(\alpha)]\widehat{e}\ .$$

PROOF. Let environment $\widehat{e}$ be defined by:

- for each variable v denoting a n-ary relation c, we define $\widehat{e}(v) = \mathsf{c}$ (the binary encoding of relation c, cf. *4.1.2*),
- for each atomic action symbol a, we define
$$\widehat{e}(a) = \{ \langle e_1', e_2' \rangle : \langle e_1, e_2 \rangle \in A(a) \},$$
where e_1', e_2' are defined from e_1 and e_2 as in Thm. 4.1.

The proof proceeds now by induction on the structure of formula α. For the simplicity we will present the proof for atomic formulas and partial correctness assertions.

Let α be atomic, i.e., $\alpha = t_1\ in\ t_2$.

$$\begin{gathered} M[t_1\ in\ t_2]e \\ = \{\text{by Thm. 4.2}\} \\ N[\mathsf{RL} \mapsto \mathsf{FRL}(t_1\ in\ t_2)]e' \\ = \{\text{because no actions occur in } \alpha\} \\ N[\mathsf{RL} \mapsto \mathsf{FRL}(t_1\ in\ t_2)]\widehat{e} \\ = \{\text{by Remark 1}\} \\ Q[\mathsf{RL} \mapsto \mathsf{FRL}(t_1\ in\ t_2)]\widehat{e} \\ = \{\text{because } \alpha \text{ is a RL formula }\} \\ Q[\mathsf{DynAlloy} \mapsto \mathsf{FDL}(t_1\ in\ t_2)]\widehat{e}\ . \end{gathered}$$

Let $\alpha = \beta\{p\}\gamma$.

$$\begin{gathered} M[\beta\{p\}\gamma]e \\ = \{\text{by def. } M\} \\ M[\beta]e \Rightarrow \forall e_1\, (\langle e, e_1 \rangle \in P[p]\ \Rightarrow\ M[\gamma]e_1) \\ = \{\text{by Thm. 4.2}\} \\ N[\mathsf{RL} \mapsto \mathsf{FRL}(\beta)]e'\ \Rightarrow \\ \forall e_1'\, (\langle e', e_1' \rangle \in P[p]\ \Rightarrow\ N[\mathsf{RL} \mapsto \mathsf{FRL}(\gamma)]e_1') \\ = \{\text{by Remark 1}\} \\ Q[\mathsf{RL} \mapsto \mathsf{FRL}(\beta)]e'\ \Rightarrow \\ \forall e_1'\, (\langle e', e_1' \rangle \in P[p]\ \Rightarrow\ Q[\mathsf{RL} \mapsto \mathsf{FRL}(\gamma)]e_1') \\ = \{\text{by semantics of FDL and definition of } \widehat{e}\} \\ Q[\mathsf{RL} \mapsto \mathsf{FRL}(\beta)\ \Rightarrow\ [p]\mathsf{RL} \mapsto \mathsf{FRL}(\gamma)]\widehat{e} \\ = \{\text{by definition of DynAlloy} \mapsto \text{FDL}\} \\ Q[\mathsf{DynAlloy} \mapsto \mathsf{FDL}\ (\beta\{p\}\gamma)]\widehat{e}\ . \end{gathered}$$

■

LEMMA 6.3. *Let α be a DynAlloy formula. Let $\widehat{e}$ be a FDL environment. Then, there exist a function A assigning meaning to actions and an environment e for DynAlloy such that*

$$Q[\mathsf{DynAlloy} \mapsto \mathsf{FDL}(\alpha)]\widehat{e} = M[\alpha]e\ .$$

PROOF. Notice that FDL environments differ from DynAlloy environments in that the former assign meaning to actions, while the latter only assign meaning to variables. Thus, from the FDL environment $\widehat{e}$ we can project a valuation e' for the variables. Notice also that e' assigns meaning to binary relations, but these relations can be seen as encodings for higher rank relations (cf. *4.1.2*). Thus, from e' we obtain the DynAlloy valuation e defined by: $e(v) = \{ \langle a_1, a_2, \ldots, a_n \rangle : \langle a_1, a_2 \star \cdots \star a_n \rangle \in e'(v) \}$. It only rests to define function A. Let a be an atomic action symbol. We define

$$A(a) = \{ \langle e_1, e_2 \rangle : \langle e_1', e_2' \rangle \in \mathsf{e}(a) \}\ .$$

The proof now proceeds by induction on the structure of the formula α and is left as an exercise for the reader. ■

THEOREM 6.4. *Let α be a DynAlloy formula. Then, α is valid in DynAlloy if and only if $\mathsf{DynAlloy} \mapsto \mathsf{FDL}(\alpha)$ is valid in FDL.*

PROOF. We will prove both implications.

$\Rightarrow$)

$$\begin{aligned} & \mathsf{DynAlloy} \mapsto \mathsf{FDL}(\alpha) \text{ is not valid in FDL} \\ \iff & \exists \widehat{e}\, (\neg Q[\mathsf{DynAlloy} \mapsto \mathsf{FDL}(\alpha)]\widehat{e}) \quad \text{(by semantics FDL)} \\ \iff & \exists e\, (\neg M[\alpha](e)) \quad \text{(by Lemma 6.3)} \\ \iff & \alpha \text{ is not valid in DynAlloy.} \quad \text{(by semantics DynAlloy)} \end{aligned}$$

$\Leftarrow$)

$$\begin{aligned} & \alpha \text{ is not valid in DynAlloy} \\ \iff & \exists e\, (\neg M[\alpha](e)) \quad \text{(by semantics DynAlloy)} \\ \iff & \exists \widehat{e}\, (\neg Q[\mathsf{DynAlloy} \mapsto \mathsf{FDL}(\alpha)]\widehat{e}) \quad \text{(by Lemma 6.2)} \\ \iff & \mathsf{DynAlloy} \mapsto \mathsf{FDL}(\alpha) \text{ is not valid in FDL .} \\ & \text{(by semantics FDL)} \end{aligned}$$

■

6.3 A Complete Calculus for FDL

In this section we present a complete calculus for FDL. Notice that due to Thm. 6.4, we can use this calculus for reasoning about the validity of DynAlloy assertions.

The set of axioms for FDL is the set of axioms for classical first-order logic, enriched with the axioms and rules for closure fork algebras, and the following axiom schemes for first-order dynamic logic:

- $\langle P \rangle \alpha \wedge [P]\beta\ \Rightarrow\ \langle P \rangle (\alpha \wedge \beta)$,
- $\langle P \rangle (\alpha \vee \beta)\ \Leftrightarrow\ \langle P \rangle \alpha \vee \langle P \rangle \beta$,
- $\langle P_0 + P_1 \rangle \alpha\ \Leftrightarrow\ \langle P_0 \rangle \alpha \vee \langle P_1 \rangle \alpha$,
- $\langle P_0; P_1 \rangle \alpha\ \Leftrightarrow\ \langle P_0 \rangle \langle P_1 \rangle \alpha$,
- $\langle \alpha ? \rangle \beta\ \Leftrightarrow\ \alpha \wedge \beta$,
- $\alpha \vee \langle P \rangle \langle P^* \rangle \alpha\ \Rightarrow\ \langle P^* \rangle \alpha$,
- $\langle P^* \rangle \alpha\ \Rightarrow\ \alpha \vee \langle P^* \rangle (\neg \alpha \wedge \langle P \rangle \alpha)$,
- $\langle x \leftarrow t \rangle \alpha\ \Leftrightarrow\ \alpha[x/t]$,
- $\alpha\ \Leftrightarrow\ \widehat{\alpha}$; 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 used for classical first-order logic plus:

- Generalization rule for the *necessarily* modal statement:

$$\frac{\alpha}{[P]\alpha}$$

- Infinitary convergence rule:

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

A proof of the completeness of the calculus for dynamic logic is presented in [18, Thm. 15.1.4]. Joining this theorem with the completeness of the axiomatization of closure fork algebras [14, Thm. 4.3], it follows that the above described calculus is complete with respect to the semantics of FDL.

7. VERIFYING FDL SPECIFICATIONS WITH *PVS*

As has been shown in Section 6, DynAlloy can be interpreted in FDL, 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 life cycle.

In Section 5.3 we showed how DynAlloy specifications can be analyzed by extending the *Alloy Analyzer* in a way it can handle partial correctness statement.

Regarding the problem of theorem proving, there are several tools that can be used to carry out this task. Among them, we can mention Isabelle [30], HOL [17], Coq [9] and *PVS* [32]. *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 [34]. Thus, we will concentrate on the use of this particular theorem prover in order to prove assertions from FDL specifications.

As it has been described in the basic *PVS* bibliography [35, 36, 37], *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 datatype specification language [33], the notion of subtypes and dependent types [37], the possibility to define parametric theories [37], and a collection of powerful proof commands to carry out propositional, equality, and arithmetic reasoning [35]. These proof commands can be combined to form proof strategies. The last feature simplifies the process of developing, debugging, maintaining, and presenting proofs.

Assertions are presented to *PVS* in the form of *sequents*. A sequent is diagrammatically presented as shown in Fig. 11.

The formulas in the upper part of the sequent are called *premises*, and those in the lower part of the sequent are the *conclusions*. A sequent as the one presented in Fig. 11 asserts that the disjunction of the conclusions follows from the conjunction of the premises. This semantics is induced by the deduction rules of the calculus.

Using *PVS* to reason about FDL specifications is not trivial because this language is not supported by the *PVS* tool

```
  {-1} alpha 1
  {-2} alpha 2
   .
   .
   .
  {-n} alpha n
|-------
  {1} beta 1
  {2} beta 2
   .
   .
   .
  {m} beta m
```

Figure 11: Diagrammatic representation of sequents.

itself. To bridge this gap, a framework was built by encoding the semantics for FDL in *PVS*' language [26].

This framework is separated in two parts. The first one is a packet of thirteen files containing all the theories needed to encode that part of the language that is common in every specification. Representative examples of these files are:

- FODL_Language.pvs: This file contains the dynamic logic language encoded as a *PVS*' abstract data-type. A commented extraction of this theory is shown in Fig. 12.

- wf_FODL_Language.pvs: In this file we define the well-formedness property which establish some restrictions on the language built in the file FODL_Language.pvs.

- FODL_semantics.pvs: This file contains the encoding of the semantics of dynamic logic. This encoding is essentially the declaration of the notion of world plus the meaning function for formulas and programs.

- FA_elements.pvs: This file contains the declaration of the base set of the algebra whose carrier set is the set of terms with which dynamic logic language is instantiated.

The other files contain those theories that depend on the specification. Taking as a case-study the memories with cache presented in section 2, we provided five files, containing the theories we used in order to build the FDL specification:

1. FA_Language.pvs: This file holds the definition of the symbols for the language of fork algebras. These are the symbols that are mandatory in the language of fork algebras such as join, meet, composition, transposition, etc., as well as those that are particular to the model under development (constant, function and predicate symbols).

2. FA_semantic.pvs: In this file we provide the definition of the semantics of the symbols of fork algebras.

3. SpecActions.pvs: In this file are defined the atomic actions required in the model.

4. SpecPredicates.pvs: Contains the definitions of the predicates that appear in the specification.

5. SpecProperties.pvs: Here are provided the assertions to be proved in the model.

```
FDL_Language[Constant: TYPE,
             Metavariable: TYPE,
             Variable: TYPE,
             Predicate: TYPE, sigPredicate: [Predicate -> nat],
             Function_: TYPE, sigFunction_: [Function_ -> nat]]:
       DATATYPE WITH SUBTYPES Term_, Formula_, Program_

  BEGIN

“Construct a Term_ from a constant / metavariable / variable symbol.”
    c(c: Constant): c?: Term_
    m(m: Metavariable): m?: Term_
    v(v: Variable): v?: Term_

“Constructs a Term_ from a function application of a function symbol to a list of Term_.”
    F(f: Function_, lF: lPrime: list[Term_] | ...): F?: Term_

“Construct a Formula_ by applying boolean operators to a / two Formula_.”
    NOT(f: Formula_): NOT?: Formula_
    OR(f_0, f_1: Formula_): OR?: Formula_

“Constructs a Formula_ from a predicate application of a predicate symbol to a list of Term_.”
    P(p: Predicate, lP: lPrime: list[Term_] | ...): P?: Formula_

“Constructs a (an equation) Formula_ from two Term_.”
    =(t_0: Term_, t_1: Term_): EQ?: Formula_

“Constructs a (universally quantified) Formula_ from (variable) Term_ and a Formula_.”
    FORALL_(x: (v?), f: Formula_): FORALL?: Formula_

“Constructs a (box) Formula_ from a Program_ and a Formula_.”
    [](P: Program_, f: Formula_): BOX?: Formula_

“Constructs a (test) Program_ from a Fomula_.”
    T?(f: Formula_): T??: Program_

“Constructs a (atomic action) Program_ from two Fomula_.”
    A(pre_post: [Formula_, Formula_]): A?: Program_

“Constructs a (skip / assignment / sequential composition / choice / iteration) Program_.”
    SKIP: SKIP?: Program_
    <|(x: (v?), t: Term_): ASSIGNMENT?: Program_
    //(P_0, P_1: Program_): COMPOSITION?: Program_
    +(P_0, P_1: Program_): CHOICE?: Program_
    *(P: Program_): ITERATION?: Program_

  END FDL_Language
```

Figure 12: Dynamic logic language encoded as a *PVS*' abstract data-type.

Once the theories are built, we can start the theorem proving process. In Figs. 13 and 14 we show, as examples, the *PVS* translation of formulas (14) and (15).

Figures 15 and 16 show the *PVS* proof scripts of the properties stated in Figs. 13 and 14.

Notice that in the proof script shown in Fig. 15 two lemmas were used to complete the proof. These lemmas state that the application of the functions SysWrite and Flush preserves the validity of the formula DirtyInv.

In FDL the lemmas are stated as follows:

$$(\forall s : \text{System}) \mid \\ (\text{DirtyInv}(s) \implies [\text{SysWrite}(s)]\text{DirtyInv}(s))$$

$$(\forall s : \text{System}) \mid \\ (\text{DirtyInv}(s) \implies [\text{Flush}(s)]\text{DirtyInv}(s))$$

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

Figure 13: *PVS* translation of Formula (14).

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

Figure 14: *PVS* translation of Formula (15).

In the proof script presented in Fig. 15, these lemmas appear referenced by the names “SysWrite_preserves_DirtyInv” and “Flush_preserves_DirtyInv”.

In the case of the proof script of Fig. 16, we also used a lemma to complete the proof. The lemma states that if the formula DirtyInv is satisfied, after the application of function DSFlush the formula FullyAgree is satisfied too. This property is specified in FDL by the formula

$$(\forall s : System)(\text{DirtyInv}(s) \Rightarrow \\ [\text{DSFlush}(s)]\text{FullyAgree}(s))\ .$$

During the proving process many strategies have been used. Some of them are strategies already defined in *PVS*, while others were implemented by us in order to make the framework friendlier to the user. Since only objects of type bool can take place in a sequent, FDL formulas cannot be part of sequents unless they are conveniently preprocessed. This is why, given a formula α, we will prove the formula

$$\textit{FORALL}\ \texttt{(w : World\_) : meaningF(f)(w)}\ . \tag{16}$$

rather than α itself. In formula (16), function *meaningF* has type $\textit{Formula\_} \to \textit{World\_} \to \textit{bool}$, and its definition is such that it asserts the validity of the formula α in the world w. Notice that there is no ambiguity in saying that α holds, because by Thm. 6.1 α is a theorem if and only if it is valid in the semantics we defined.

In order to improve readability of formulas (and therefore the usability of the tool), we have defined a conversion so that the user of the framework can simply declare

```
Theorem_1 : THEOREM f
```

which is automatically turned into

```
Theorem_1 : THEOREM
        FORALL (w : World_) : meaningF(f)(w) .
```

This means that whenever the user attempts to prove a theorem declared as “`Theorem_1 : THEOREM f`”, *PVS* internally builds the sequent

```
|-------
{1} FORALL (w : World_) : meaningF(f)(w)
```

Notice that there is no harm or ambiguity in pretty-printing the sequent as

```
;;; Proof for formula SpecProperties.Preservation_of_DirtyInv ;;;
developed with old decision procedures (""
 (EXPAND-MEANING)
 (EXPAND-MEANING 1)
 (EXPAND-MEANING 1)
 (SKOSIMP*)
 (PURIFY-FODL -1)
 (LEMMA "PDL_6_box_form")
 (INST -1 "DirtyInv(v(cs))"
  "SysWrite(v(cs)) + Flush(v(cs))"
  "w!1 WITH [(cs) := t!1]")
 (EXPAND-MEANING -1)
 (INST?)
 (EXPAND-MEANING -1)
 (PROP)
 (HIDE 2)
 (EXPAND-MEANING 1)
 (PROP)
 (("1" (PURIFY-FODL 1))
  ("2"
   (EXPAND-MEANING 1)
   (SKOSIMP*)
   (EXPAND-MEANING 1)
   (PROP)
   (PURIFY-FODL -2)
   (EXPAND-MEANING 1)
   (SKOSIMP*)
   (PURIFY-FODL 1)
   (HIDE -1 -4)
   (PURIFY-FODL -2)
   (PROP)
   (("1"
     (LEMMA "SysWrite_preserves_DirtyInv")
     (PURIFY-FODL -1)
     (INST -1 "wPrime!2")
     (INST -1 "mMetavariable!1")
     (INST -1 "wPrime!1(cs)")
     (PROP)
     (INST -1 "wPrime!2")
     (PROP))
    ("2"
     (LEMMA "Flush_preserves_DirtyInv")
     (PURIFY-FODL -1)
     (INST -1 "wPrime!2")
     (INST -1 "mMetavariable!1")
     (INST -1 "wPrime!1(cs)")
     (PROP)
     (INST -1 "wPrime!2")
     (PROP))))))
```

Figure 15: *PVS* proof script of formula in Fig. 13.

```
|-------
{1} FORALL (w : World_) : (f)(w)
```

because constructing the semantics of f and proving that the formula describing the semantics holds in every world is the only way to prove, in *PVS*, that f is a theorem. Thus, the application of the function *meaningF* can (and most often will) remain implicit. In order to leave the application implicit we built a strategy in *PVS* that unfolds the meaning function but avoids making any explicit reference to *meaningF* in the resulting expression. For instance, if we unfold the (implicit) occurrence of *meaningF* in the formula $(\alpha \vee \beta)(w)$, we will obtain the formula $(\alpha)(w) \vee (\beta)(w)$.

As it is shown in the proof script presented in Fig. 15, the first strategy applied is (EXPAND-MEANING) and the result is the sequent

```
|-------
{1} FORALL (w:World_):
        FORALL (mMetavariable:AssMetavariable):
         (f)(mMetavariable)(w)
```

```
;;; Proof for formula SpecProperties.Consistency_criterion ;;;
developed with old decision procedures (""
 (EXPAND "Consistency_criterion" 1)
 (EXPAND-MEANING 1)
 (EXPAND-MEANING 1)
 (EXPAND-MEANING 1)
 (SKOSIMP*)
 (PURIFY-FODL -1)
 (LEMMA "PDL_4_box_form")
 (INST -1 "FullyAgree(v(cs))"
  "*(SysWrite(v(cs)) + Flush(v(cs)))"
  "DSFlush(v(cs))" "w!1 WITH [(cs) := t!1]")
 (EXPAND-MEANING -1)
 (INST -1 "mMetavariable!1")
 (EXPAND-MEANING -1)
 (PROP)
 (HIDE 2 3)
 (EXPAND-MEANING 1)
 (SKOSIMP*)
 (LEMMA "Preservation_of_DirtyInv")
 (EXPAND "Preservation_DirtyInv" -1)
 (EXPAND-MEANING -1)
 (EXPAND-MEANING -1)
 (INST -1 "w!1 WITH [(cs) := t!1]")
 (INST -1 "mMetavariable!1")
 (INST -1 "(w!1 WITH [(cs) := t!1])(cs)")
 (EXPAND-MEANING -1)
 (PROP)
 (("1"
   (EXPAND-MEANING -1)
   (INST -1 "wPrime!1")
   (PROP)
   (PURIFY-FODL -1)
   (LEMMA "DSFlush_leaves_FullyAgree")
   (EXPAND "DSFlush_leaves_FullyAgree" -1)
   (EXPAND-MEANING -1)
   (EXPAND-MEANING -1)
   (INST -1 "wPrime!1")
   (INST -1 "mMetavariable!1")
   (INST -1 "wPrime!1(cs)")
   (EXPAND-MEANING -1)
   (PROP)
   (("1" (HIDE -2 -3 -4) (PURIFY-FODL))
    ("2" (HIDE -2 -3 2) (PURIFY-FODL))))
  ("2" (HIDE -1 2) (PURIFY-FODL))))
```

Figure 16: *PVS* proof script of the formula in Fig. 14.

This sequent is the pretty-printed version of the sequent

```
|-------
{1} FORALL (w:World_):
        FORALL (mMetavariable:AssMetavariable):
          m(mMetavariable)(inl(f)) w
```

and considers a new definition of the meaning function that involves the use of valuations for rigid variables. These valuations are necessary for the sake of specifying pre and post conditions. The next example shows how this rigid variables are used:

$$\begin{aligned}&(\forall m : \text{Memory})(\forall d : \text{Data})(\forall a : \text{Addr}) \mid \\ &\quad ((m = M_0 \ \wedge \ d = D_0) \implies \\ &\qquad [\text{Write}(m, a, d); \text{Read}(m, a, d)](d = D_0))\end{aligned}$$

In the previous assertion, M_0 and D_0 are variables that record the initial values of variables m and d, respectively. The value of M_0 and D_0 must remain the same in all worlds, and this is the reason why these variables are called *rigid*. In order to prove the validity of the assertion it is necessary to allow M_0 and D_0 range over all possible memories and data, respectively. Since `mMetavariable` ranges over valuations for the rigid variables, this is achieved by universally quantifying `mMetavariable`. From now on the meaning function will be denoted by `m`, and will recursively construct the semantics of a formula each time its definition is implicitly expanded by the application of the strategy (EXPAND-MEANING ...) to a formula number.

The next strategy applied in the proof script is called SKOSIMP. This strategy skolemizes a universal quantifier, and the star is used to tell *PVS* that it should skolemize as many quantifiers as possible, even if that requires simplifying the sequent by breaking conjunctions and disjunctions in the sequent. Essentially, if we apply this strategy to the sequent

```
|-------
{1} FORALL (w: World_):
        FORALL (mMetavariable: AssMetavariable):
          (f)(mMetavariable)(w)
```

we will obtain as a result

```
|-------
{1} (f)(mMetavariable!1)(w!1)
```

due to the introduction of Skolem constants to replace the quantifiers.

After that, a strategy called PURIFY-FODL is applied. This strategy was designed to perform all the expansions necessary in order to construct the semantics of the formula whose number is given as argument. If the argument is omitted, all the formulas in the sequent are expanded. Notice that this procedure involves the recursive expansion of the meaning function.

The use of the command LEMMA allows the user to introduce a given formula as a hypothesis (it will appear in the upper part of the sequent, and will be numbered as -1). To get a complete proof of the target property this formula must be discharged. Otherwise, the proof is considered incomplete because it relays on a lemma whose proof is still pending. Suppose we want to use a formula g, that was named "hypothesis_for_f" when it was declared, as an assumption to prove formula f. Applying the command (LEMMA "hypothesis_for_f") has the following effect:

```
{-1} g
|-------
{1} (f)(mMetavariable!1)(w!1)
```

Following the proof script, the INST command is applied. This command tells *PVS* that the universal quantifiers in the formula given as argument must be instantiated with the terms listed in the call. The quantifiers are instantiated in the order they appear in the formula. Notice that one of the terms used to instantiate the quantifiers is "`w!1 WITH [(cs) := t!1]`". This is the *PVS* notation for functional update, and stands for the world (valuation) that agrees in all variables but `cs` with world `w!1`. This world evaluates variable `cs` to the value "`t!1`".

Another command used during the proof is HIDE. This command hides formulas that appear in a sequent, therefore improving readability. If we have the sequent

```
{-1} f1
{-2} f2
{-3} f3
|-------
{1} g1
{2} g2
```

and apply (HIDE -2 2), the result is the sequent

```
{-1} f1
{-2} f3
|-------
{1} g1
```

The command EXPAND appearing in the proof script is a primitive *PVS* command that allows one to substitute an identifier by its definition. For instance, if function g is defined by $g(x,y) = x+y$, after the application of (EXPAND "g" 1) to the sequent

```
|-------
{1} f (x!1, y!1) = g (r!1, s!1)
```

we obtain the sequent

```
|-------
{1} f (x!1, y!1) = r!1 + s!1
```

The last command to which we make reference in the proof script is PROP. This command is used in order to simplify a sequent
by:

- splitting conjunctions in the thesis part of the sequent or disjunctions in the hypothesis part,
- flattening disjunctions in the thesis part of the sequent or disjunctions in the hypothesis part.

The effect of this command can be seen as follows:

-
  ```
  |-------        is transformed to
  {1} p AND q

       |-------   and  |-------
       {1} p           {1} q
  ```

-
  ```
  {1} p AND q
  |-------        is transformed to

       {-1} p
       {-2} q
       |-------
  ```

-
  ```
  |-------        is transformed to
  {1} p OR q

       |-------
       {1} p
       {2} q
  ```

-
  ```
  {-1} p OR q
  |-------        is transformed to

      {-1} p          {-1} q
      |-------   and  |-------
  ```

Notice that the remaining strategies and commands used in the proofs are among the ones explained above. Even if the proof seems to be cryptic for readers not familiar with *PVS*, it is quite short and straightforward for users used to the framework and *PVS*' language. This is in part because the use of lemmas simplifies the process of proving a property by allowing modular proofs. We recommend the reading of [26] as the reference material for this section.

8. CONCLUSIONS AND FURTHER WORK

We have succeeded in finding a logic that can be understood by an Alloy user without demanding significant new skills. This logic possesses a complete and purely relational equational calculus which can be used in the verification of Alloy assertions. Further work includes the inclusion of the calculus in an axiomatic theorem prover such as, for instance, Isabelle [30]. Also, we presented in Section 4.1.4 an example of a property that cannot be analyzed with the Alloy Analyzer but can be analyzed in FRL. This has to be studied further in order to determine to what extent it extends to other assertions.

Extending Alloy with actions allowed us to deal with properties of executions in a more natural and abstract way. We are currently modifying the Alloy analyzer's source code in order to analyze properties involving actions.

9. ACKNOWLEDGEMENTS

We wish to thank Daniel Jackson for reading preliminary versions of this paper and making valuable suggestions. Also Sam Owre and Natarajan Shankar are to be thanked for their work on the verification of properties using PVS.

10. REFERENCES

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

[2] Arkoudas, K, *Denotational Proof Languages*, PhD thesis, MIT, 2000.

[3] Arkoudas K., Khurshid S., Marinov D. and Rinard M., *Integrating Model Checking and Theorem Proving for Relational Reasoning*, to appear in Proceedings of RelMiCS'03.

[4] Bickford M. and Guaspari D., *Lightweight Analysis of UML*. TM-98-0036, Odyssey Research Associates, Ithaca, NY, November 1998.

[5] Booch G., Jacobson I. and Rumbaugh J., *The Unified Modeling Language User Guide*, The Addison-Wesley Object Technology Series, 1998.

[6] Burris, S. and Sankappanavar, H.P., *A Course in Universal Algebra*, Graduate Texts in Mathematics 78, Springer–Verlag, 1981.

[7] E. Clarke, O. Grumberg and D. Peled, *Model Checking*, The MIT Press, 2000.

[8] R. Cleaveland, M. Klein and B. Steffen, *Faster Model Checking for the Modal Mu-Calculus*, in Proceedings of the Fourth International Workshop on Computer Aided Verification, LNCS, Springer-Verlag, 1992.

[9] Coq Development Team, *The Coq Proof Assistant - Reference Manual*, Institut National de Recherche en Informatique et en Automatique (INRIA), 2001.

[10] Dijkstra E. W. and Scholten C. S., *Predicate calculus and program semantics*, Springer-Verlag, 1990.

[11] 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.

[12] 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.

[13] 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.

[14] Frias M., *Fork Algebras in Algebra, Logic and Computer Science*, World Scientific Publishing Co., Series Advances on Logic, 2002.

[15] Frias M.F., Baum G.A. and Maibaum T.S.E., *Interpretability of First-Order Dynamic Logic in a Relational Calculus*, in Proceedings of RelMiCS 6, LNCS 2561, Springer-Verlag, 2002.

[16] Frias M., López Pombo C., Baum G., Aguirre N. and Maibaum T., *Taking Alloy to the Movies*, in Proceedings of FME'03, Pisa, Italy, 2003, LNCS 2805, pp. 678–697.

[17] Gordon M., and Melham T.,*Introduction to HOL: A theorem proving environment for higher order logic*, Cambridge University Press, 1993.

[18] Harel D., Kozen D. and Tiuryn J., *Dynamic Logic*, MIT Press, October 2000.

[19] Jackson D., *Automating First-Order Relational Logic*, in Proceedings of SIGSOFT FSE 2000, pp. 130-139, Proc. ACM SIGSOFT Conf. Foundations of Software Engineering. San Diego, November 2000.

[20] Jackson D., *Micromodels of Software: Lightweight Modelling and Analysis with Alloy*, 2002.

[21] 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.

[22] Jackson D. and Sullivan K., *COM Revisited: Tool Assisted Modelling and Analysis of Software Structures*,

[23] Jackson D., Schechter I. and Shlyakhter I., *Alcoa: the Alloy Constraint Analyzer*, Proceedings of the International Conference on Software Engineering, Limerick, Ireland, June 2000.

[24] 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.

[25] Jones C.B., *Systematic Software Development Using VDM*, Prentice Hall, 1995.

[26] 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.

[27] Maddux R., *Pair-Dense Relation Algebras*, Transactions of the American Mathematical Society, Vol. 328, N. 1, 1991.

[28] 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.

[29] K. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[30] Nipkow T., Paulson L. C. and Wenzel M., *A Proof Assistant for Higher-Order Logic*, Springer Verlag, 1st. edition, March 2002.

[31] *Object Constraint Language Specification". Version 1.1*, 1 September 1997.

[32] 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.

[33] Owre S. and Shankar N., *Abstract datatypes in PVS*, Technical Report CSL-93-9R, SRI International, December 1993. Subtantially revised in June 1997.

[34] 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.

[35] Owre S., Shankar N., Rushby J,M, and Stringer-Calvert D.W.J., *PVS Prover Guide*, SRI International, version 2.4 edition, November 2001.

[36] Owre S., Shankar N., Rushby J.M. and Stringer-Calvert D.W.J., *PVS System Guide*, SRI International, version 2.4 edition, December 2001.

[37] Owre S., Shankar N., Rushby J.M. and Stringer-Calvert D.W.J., *PVS Language reference*, SRI International, version 2.4 edition, December 2001.

[38] Spivey J.M., *Understanding Z: A Specification Language and Its Formal Semantics*, Cambridge Tracts in Theoretical Computer Science, 1988.

[39] Tarski, A. and Givant, S.,*A Formalization of Set Theory without Variables*, A.M.S. Coll. Pub., vol. 41, 1987.

[40] M. Vardi and P. Wolper, *An automata-theoretic approach to automatic program verification*, in Proceedings of the 1st IEEE Symposium on Logic in Computer Science, 1986, pp. 322-331.