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

MARCELO F. FRIAS and CARLOS G. LÓPEZ POMBO
FCEyN, Universidad de Buenos Aires and CONICET
GABRIEL A. BAUM
Facultad de Informática, Universidad Nacional de La Plata and CONICET
NAZARENO M. AGUIRRE
FCEFQyN, Universidad Nacional de Río Cuarto and CONICET
and
THOMAS S. E. MAIBAUM
Department of Computing & Software, McMaster University

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 analyzing) 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 characterization of executions.

We propose to overcome the first problem by translating relational logic to the equational calculus of *fork algebras*. Fork algebras provide a purely relational formalism close to Alloy, which

Marcelo F. Frias wishes to thank the Antorchas Foundation.
Authors' addresses: M. F. Frias and C. G. López Pombo, Department of Computer Science, School of Sciences, University of Buenos Aires, Buenos Aires, Argentina (1427); email: {mfrias, clpombo}@dc.uba.ar; G. A. Baum, Facultad de Ciencias Exactas, Universidad Nacional de La Plata, Calle 50 y 115- ler piso (1900), La Plata, Argentina; email: gbaum@sol.info.unlp.edu.ar; N. M. Aguirre, Departamento de Computación, FCFQyN, Universidad de Rio Cuarto, Enlace rutas 8y36, Km 601, Rio Cuarto (5800), Córdobu, Argentina; email: aguirre@dc.exa.unrc.edu.ar; T. S. E. Maibaum, Department of Computing and Software, McMaster University, 1280 Main Street West, Hamilton, ON, Canada L8S 4L7; email: tom@maibaum.org.

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 them to state properties of execution traces at an appropriate level of abstraction.

Since automatic analysis is one of Alloy's main features, and this article 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,

—the complete calculus for the relational logic is straightforwardly extended to a complete calculus for the extension of Alloy.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods*

General Terms: Design, Verification

Additional Key Words and Phrases: Alloy, fork algebras, relational specifications

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. *UML* [Booch et al. 1998] 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. 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* [Abrial 1996], *VDM* [Jones 1986], *Z* [Spivey 1988], and Alloy [Jackson et al. 2001].

Formal semantics is not necessarily enough for making specifications analyzable: 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 analyzability 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's 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, there 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 [Jackson 2000, p. 1]."

This analysis has been implemented by the Alloy Analyzer [Jackson et al. 2000], 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 analyze 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 [Jackson and Sullivan 2000].

1.1 Contributions of this Article

The contributions of this article are twofold. First, we 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 [Frias 2002] 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 does not compromise the analyzability of specifications. In fact, resorting to FRL enables us to strictly analyze more properties than with the Alloy Analyzer under the standard Alloy semantics [Jackson et al. 2000].

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 [Jackson et al. 2001], their proposed solution requires a complicated and unstructured characterization 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 schemas 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 article), 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 Jackson et al. [2001, Section 2.6]) for modeling the 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 article is providing a complete calculus for Alloy, it is worthwhile asking ourselves whether the deductive calculus for the relational logic presented in this article can be extended to DynAlloy. We show that by 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 also perform deductive reasoning 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 Jackson [2002b, 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* [Owre et al. 1998].

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

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

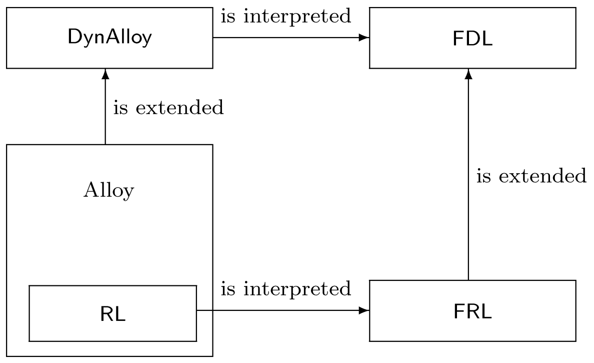

Fig. 1. Relationships among the formalisms.

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. [2004] on their tool Prioni, presented some time after the first submission of this article at RelMiCS'03.[1] Prioni uses a semi-automatic theorem prover called Athena [Arkoudas 2000] in order to prove properties regarding Alloy specifications. Arkoudas et al.'s characterization of Alloy allows one to reason about specifications, using semi-automatic theorem proving. However, the proposed characterization 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 [Arkoudas et al. 2004, p. 6]."

Prioni also has a number of further shortcomings, such as the, in our opinion, awkward representations of Alloy's composition operation '.' and of ordered pairs [Arkoudas et al. 2004, p. 4]. This tool, however, succeeds in integrating 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 characterization 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 article.

The deduction system we propose is based on purely relational (i.e., only the type of relations is present) calculi for FRL and FDL, which are complete with

[1]Note that our approach is previous to Arkoudas et al.'s: Arkoudas et al. [2004] includes a reference to a preliminary version of this article [Frias et al. 2003].

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 characterization 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* [Clarke et al. 2000]. 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 does or does not satisfy a logical property. There exist various approaches to efficient model checking, such as the automata-theoretic [Vardi and Wolper 1986], the semantic [Cleaveland et al. 1993], and the symbolic ones [McMillan 1993].

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 Jackson [2002a]. Nevertheless, model checking techniques might be useful for the verification of properties regarding traces of executions of Alloy specifications. In fact, as demonstrated in Jackson et al. [2001], there exists an interest in describing and analyzing the possible behaviors of systems specified in Alloy; furthermore, as we said, it is one of our aims to contribute to a better characterization of these behaviors. The description of executions proposed in Jackson et al. [2001], by introducing system traces, clock ticks, and so on, *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 characterization of behaviors would complicate the possibility of applying model checking techniques to verify properties of executions. Our approach, 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 organized and simple characterization 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 article.

1.3 Structure of the Article

The remainder of this article is organized as follows. In Section 2, we present a description of the syntax and semantics of the current version of standard Alloy, as presented in Jackson et al. [2001]. 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 Jackson et al. [2001]. 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 systems involving memories with cache. We might recognize 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 a set of addresses, and a total mapping from these addresses to data values:

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

The symbol "!" in this 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 another "parent" signature. This is done via what is called *signature extension*. For 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 recognized as *dirty*.

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

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

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

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

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

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

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

Fig. 2. Grammar and semantics of Alloy.

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 fields of a signature denote *relations*. For instance, the "addrs" field 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 Figure 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 Jackson [2002a], 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:

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

Operations for transitive closure and transposition are only defined for binary relations. Thus, function X in Figure 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 spaces 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)
}
```

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 from the application of function Write, a -> d denotes the ordered pair ⟨a, d⟩, and ++ denotes relational overriding, defined as follows[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: 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

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

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 Jackson et al. [2001], 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 characterize 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 [Object Management Group 1997]. Properties or constraints in Alloy are defined as *facts*. To give an idea of how constraints or properties are specified, we reproduce some here. 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 in 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.

3. FEATURES AND DEFICIENCIES OF ALLOY

In this section, we summarize 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* [Spivey 1988], *VDM* [Jones 1986] or *B* [Abrial 1996], Alloy's semantics is strongly based on the use of *relations*. A main distinguishing characteristic of Alloy, that we mentioned before in this article, is that it has been designed with the goal of making specifications automatically analyzable. 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 Jackson [2002a], are the following:

—Fulfilling the goal of an analyzable language made Alloy a simple language, with a clear and elegant semantics based on relations.
—Regardless of its simplicity, Alloy supports some constructs that resemble common idioms of object modeling. Perhaps this feature is one of the main reasons 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) [Object Management Group 1997]. The well defined and concise syntax of Alloy is much easier to understand than the, in our opinion, rather cumbersome OCL grammar, as presented in Object Management

Group [1997]. 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 [Bickford and Guaspari 1998].

—The syntax of Alloy, which includes both textual and graphical notations, is based on a small underlying formalism, RL, with few constructs. The relational semantics of RL allows one to refer to relations, sets, and individual atoms with the same simplicity.

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

—Alloy was designed with the goal of making specifications automatically analyzable 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 semiautomated deduction can be used successfully, especially in combination with fully automatic analysis. The Stanford Temporal Prover (STeP) [Manna et al. 1994], for instance, is a good example of a tool combining, with great success, fully automatic verification (in this case, model checking) with semiautomated deduction.

Providing Alloy with theorem proving is not a particularly complicated task. As we mentioned, Arkoudas et al. [2004] 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, if one counts on a complete proof system for the logic, then all the statements expressible in the logic that are consequences of the *axioms* of a specification are provable.

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.

Jackson et al. [2001] present a methodology for checking properties of executions in Alloy. The method presented 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, and so on. 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 a 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 organized manner. DynAlloy supports the description of assertions regarding executions. DynAlloy can then be interpreted over a dynamic logic extending FRL. An 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 such 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
>
> $$some\ s : state \mid op1(pre, s)\ and\ op2(s, post)$$
>
> but this calls for a second-order quantifier." [Jackson 2002a, Section 6.2]

A partial solution to this problem was proposed in [Jackson et al. 2001], 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 regarding Alloy, the semantics of Alloy's relational logic is defined in terms of binary relations. The current semantics [Jackson et al. 2001] 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) Some of Alloy's 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 our framework of binary relations.

4.1 Closure Fork Algebras

Fork algebras [Frias 2002] are described through a 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 with respect to $B \times B$, 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:

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

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

The class of proper closure fork algebras can be characterized by a set of formulas and inference rules. These constitute what we call *Fork Relational Logic*, denoted by FRL.

The axioms of FRL are composed of:

(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. We denote by $\leq$ the ordering induced by the Boolean axioms, and defined by $x \leq y \iff x \cup y = y$.
(2) Formulas defining composition of binary relations, transposition and the identity relation:

$$x;(y;z) = (x;y);z,$$
$$x;Id = Id;x = x,$$
$$(x;y) \cap z = \emptyset \text{ iff } (z;\breve{y}) \cap x = \emptyset \text{ iff } (\breve{x};z) \cap y = \emptyset.$$

(3) Formulas defining the operator ∇:

$$x \nabla y = (x;(Id \nabla 1)) \cap (y;(1 \nabla Id)),$$
$$(x \nabla y);(w \nabla z)\breve{} = (x;\breve{w}) \cap (y;\breve{z}),$$
$$\left(Id \nabla 1\right)\breve{}\,\nabla \left(1 \nabla Id\right)\breve{} \leq Id.$$

(4) Formulas axiomatizing reflexive-transitive closure:

$$x^* = Id \cup (x;x^*),$$
$$x^*;y;1 \leq (y;1) \cup (x^*;(\overline{y;1} \cap (x;y;1))).$$

The inference rules for FRL are those for equational logic (see for instance Burris and Sankappanavar [1981, p. 94]), plus the following equational (but infinitary) proof rule for reflexive-transitive closure[3]:

$$\frac{\vdash Id \leq y \qquad x^i \leq y \vdash x^{i+1} \leq y}{\vdash x^* \leq y}.$$

The axioms and rules given above define a class of models. It is relatively straightforward to prove that proper closure fork algebras are among these models, since they satisfy the axioms [Frias et al. 1997]. It could also be the case that there are models of the axioms that are not proper closure fork algebras; this would mean that the above axioms and inference rules do not *precisely* characterize proper closure fork algebras. Fortunately, as it was proved in Frias et al. [2002] (which heavily relies on Frias et al. [1997]), if a model is not a proper closure fork algebra then it is isomorphic to one.

It is also worth noting that in fork algebras, binary relations are first-order citizens. 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 us 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 Figure 3. The set U is the domain of a proper fork algebra, and therefore a set of binary relations.

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

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

Fig. 3. Semantics of fork terms and equations involving variables.

4.1.1 *Representing Objects and Sets.* Sets will be represented by partial identities; that is, binary relations contained in the identity relation *Id*. Thus, for an arbitrary type t, the *meaning* of t within an environment *env*, denoted by $env(t)$, must be a partial identity. Variables of type t are represented, within an environment *env*, by singletons of the form $\{ \langle x, x \rangle \}$, where $\langle x, x \rangle \in env(t)$. This can be characterized by the following conditions[4]:

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

In fact, it is not difficult to prove that, given binary relations x and y satisfying the properties:

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

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 $\{ \langle a, a \rangle \}$. For relations x and y, we denote by $x : y$ the fact that x is of type y; that is, that x and y satisfy the formulas in (2).

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

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

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

$$\begin{array}{l}
\pi = \{ \langle a \star b, a \rangle : a, b \in B \}, \\
\rho = \{ \langle a \star b, b \rangle : a, b \in B \}.
\end{array}$$

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

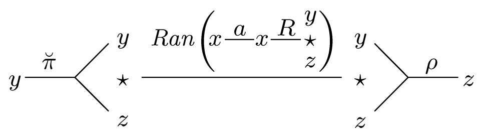
y
$\breve{\pi}$
y
$\star$
z
$Ran\left(x \xrightarrow{a} x \xrightarrow{R} \begin{smallmatrix} y \\ \star \\ z \end{smallmatrix}\right)$
y
$\star$
z
ρ
z

Fig. 4. Semantics of $\bullet$.

Given an 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:

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

In order to clarify our treatment of n-ary relations, let us consider a small example. Recalling signature *Memory*, field map stands in Alloy for a ternary relation

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

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. In general, we will denote by C the encoding of an n-ary relation C as a binary relation.

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

$$Ran(x) = \left(x ; 1\right) \cdot Id.$$

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 $x \xrightarrow{R} y$, the fact that x and y are related via the relation R, then Figure 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

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

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

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

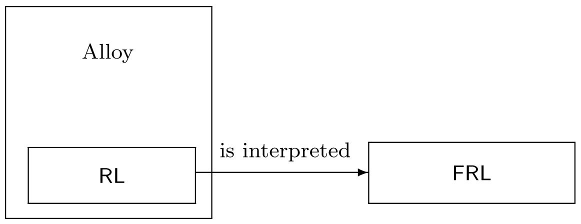

Fig. 5. Relationships among the formalisms.

In case relation R is not a set—it represents a relation whose rank is greater than or equal to 2—navigation R • S can still be defined. Nevertheless, its relational definition in terms of binary relations is more cumbersome. Since in object oriented settings navigation usually falls in the situation where relation R is a set, we will not deal here with the general case.

4.1.3 *Translating Alloy Formulas into Fork Algebra Equations.* A problem that is central to our work is that of interpreting RL, and consequently Alloy as a whole, into the FRL. In this section, we deal with this problem, and show how RL formulas can be interpreted as FRL equations. If we want, as we propose, to use FRL as a proof system for (extended) Alloy specifications, the translation of an RL formula α into a set α_∇ of FRL equations must be done in such a way that α is *valid* if and only if α_∇ is *provable* in FRL. Regarding Figure 1, in this section we deal with the portion depicted in Figure 5.

Atomic RL formulas are straightforwardly translated into FRL, since they can be seen as equations. Dealing with atomic formulas is not sufficient, since we still need to deal with Boolean connectives and quantifiers. The case of Boolean connectives does not constitute a problem: As we will show next, Boolean combinations of RL equations can be reduced to a single RL equation, and therefore can be easily translated into FRL. Formulas involving quantification are then the remaining problem. We will show how to handle these at the end of this section.

This section is rather theoretical, and some readers might want to skip it; the remaining sections can be understood without going into the details of how Alloy formulas are represented in terms of fork algebra equations.

It is well known [Tarski and Givant 1987, 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:

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

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:

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

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,

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

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 $x_1, \ldots, x_m$. Since variables in RL stand for single objects, instantiating $x_1, \ldots, x_m$ with atoms $b_1, \ldots, b_m$ yields a n-ary relation denoted by $t(b_1, \ldots, b_m)$. In this section, term t will be translated to a term $T_m(t)$ such that

$$\langle x, y\rangle \in t(b_1, \ldots, b_m) \iff \langle (b_1 \star \cdots \star b_m) \star x, (b_1 \star \cdots \star b_m) \star y\rangle \in T_m(t).$$

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

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

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:

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

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:

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

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

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

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

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,

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

In the following theorem we assume that, whenever the transpose operation or the transitive closure occur in a term, they affect a binary relation. Notice that this very same assumption is also made in Jackson et al. [2001]. We also assume that whenever the navigation operation is applied, the argument on the left-hand side is a unary relation (set). This is because our representation of relations of arity greater than two makes defining the generalized composition more complicated than desirable. At the same time, the use of navigation in object-oriented settings usually falls in this situation. 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 the 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 an n-ary relation,*
(2) *there are m free variables* $x_1, \dots, x_m$ *in t,*

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

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.

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

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

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

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

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

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

$$\begin{aligned} e'(T_m(v)) &= e'((Id_{S_1} \otimes \cdots \otimes Id_{S_m}) \otimes v) \\ &\qquad \text{(def. } T_m) \\ &= (Id_{S_1} \otimes \cdots \otimes Id_{S_m}) \otimes e'(v) \\ &\qquad \text{(semantics)} \\ &= (Id_{S_1} \otimes \cdots \otimes Id_{S_m}) \otimes \{\langle a_1, a_2 \star \cdots \star a_n\rangle : \langle a_1, \dots, a_n\rangle \in e(v)\} \\ &\qquad \text{(def. } e') \\ &= \{\langle b^\star \star a_1, b^\star \star (a_2 \star \cdots \star a_n)\rangle : \langle a_1, \dots, a_n\rangle \in e(v)\} \\ &\qquad \text{(semantics)} \\ &= \{\langle b^\star \star a_1, b^\star \star (a_2 \star \cdots \star a_n)\rangle \;: \langle a_1, \dots, a_n\rangle \in (e(\overline{b} \mapsto \overline{x})))(v)\} \\ &\qquad \text{(def. } e(\overline{b} \mapsto \overline{x})) \end{aligned}$$

□

In order to translate an 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:

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

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.

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 RL $\mapsto$ FRL (mapping RL sentences to FRL equations) on a sentence α in the following way:

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

atomic formula: 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 Theorem 4.1, $T_m(t)$ is a binary relation whose elements are pairs of the form

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

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

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

Formula α states that every n-tuple belongs to the semantics of t. Therefore, we must universally quantify 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

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

[5]Given a binary relation R, $\mathsf{ran}(R)$ denotes the set $\{b : \exists a$ 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.

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. Taking advantage of 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 $(\forall_{x_1})\cdots(\forall_{x_n})\dagger(t)$.

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 $T'_m(\alpha) = t_1 \cap t_2$.

disjunction: Let $\alpha = \beta \mid\mid \gamma$. 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 $\alpha =$ 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 $\alpha =$ 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,*

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

where environment e' is defined as in Theorem 4.1.

Example. Let us consider the following assertion:

$$\text{some s} : \textit{System} \mid \text{ s.cache.map in s.main.map.} \tag{4}$$

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

$$\text{some s} : \textit{System} \mid (1_2 - \text{ s.cache.map}) + \text{ s.main.map} = 1_2. \tag{5}$$

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

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

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

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

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

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

which we can try to prove by equational reasoning within 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 an RL formula α whose FRL translation is a fork equation e_α, we can compute the semantics of e_α using function N (cf. Figure 3). Thus, an immediate question one might ask 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 whether 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 $\bigcup_{0 \leq i} D^\star_i$, where

$$\begin{aligned} D^\star_0 &= D, \\ D^\star_{n+1} &= D^\star_n \cup \{a \star b : a, b \in D^\star_n\}. \end{aligned}$$

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, Theorem 4.2 shows that

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

where environment e' is defined as in Theorem 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 the above recipe relies on the use of the Alloy tool. An alternative approach would be to translate a FRL specification into a SAT problem, and then make use of the SAT solvers on the propositional formulas thus obtained. When translating an Alloy specification, a field R in a specification, standing for a n-ary relation on $A_1 \times \cdots \times A_n$, can be seen as a bit n-dimensional matrix M. If we denote by a^i_j the j-th atom from domain A_i, M will hold a 1 in position $[x_1, \ldots, x_n]$ $(0 \leq x_i < |A_i|)$ when the tuple of atoms $\langle a^1_{x_1}, \ldots, a^n_{x_n} \rangle$ belongs to R. The translation [Jackson 2000] then associates a propositional variable $p_{x_1,\ldots,x_n}$ to each position in M. When translating FRL, fields always stand for binary relations. Nevertheless, since fork induces tupling of atoms, a field will stand for a binary relation holding pairs of the form $\langle a_1 \star \cdots \star a_m, b_1 \star \cdots \star b_n \rangle \in (A_1 \times \cdots \times A_m) \times (B_1 \times \cdots \times B_n)$. If atom a_i is the x_i-th atom in A_i $(1 \leq i \leq m)$ and atom b_j is the y_j-th atom in B_j $(1 \leq j \leq n)$, we associate a propositional variable $p_{\langle x_1,\ldots,x_m\rangle,\langle y_1,\ldots,y_n\rangle}$. The translation then proceeds as with Alloy specifications (with minor modifications regarding handling of indices), with the exception of the new operator fork. Let $T = R \nabla S$. Let R, S and T be the bidimensional bit matrices associated to the relations R, S and T, respectively. If we denote by $r_{i,j}$, $s_{i,j}$, $t_{i,j}$ the propositional variable associated to position $[i, j]$ of R, S, T, respectively, we establish the correspondence:

$$t_{i,\langle j,k\rangle} = r_{i,j} \wedge s_{i,k}.$$

Notice that in Theorem 4.2 we assume 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. This is because counterexamples in Alloy are built from finite relations, while elements in a non proper fork algebra may represent infinite relations. In the next paragraphs we will discuss this briefly. A complete discussion exceeds the scope of this article.

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. The specification of function Flush in Section 2.1 has the following form:

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

This is recognized within Alloy as a higher order formula [Jackson 2002b] because quantification in Alloy is only defined over atoms, yet x takes as values arbitrary unary relations (sets). Let us analyze what happens in the modified semantics. Since t is a type (set), it stands for a subset of Id. Similarly, subsets of t are subsets of the identity, which are contained in t. Thus, formula (8) is an abbreviation for

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

which is a first-order formula on the language of fork algebras.

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

$$some\ s : state \mid op1(pre, s)\ and\ op2(s, post) \tag{9}$$

is first-order with the modified semantics. Operations $op1$ and $op2$ are nothing but binary predicate symbols added to the first-order language of fork algebras, and thus formula (9) is first-order.

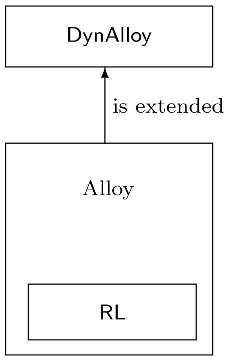

Fig. 6. Alloy and its dynamic extension DynAlloy.

This result shows that the newly defined semantics fits in a better way with the language, compared with the standard semantics.

5. ADDING DYNAMIC FEATURES TO ALLOY

In this section we start with the second part of our contribution. We extend Alloy's relational logic syntax and semantics with the aim of dealing with properties of executions of operations specified in Alloy. Recalling Figure 1, in this section we will deal with the portion reproduced in Figure 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 states (i.e., they describe relations between input and output data). Actions are built out of 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 [Harel et al. 2000], and the fact that dynamic logic is suitable for expressing partial 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 [Dijkstra and Scholten 1990]. 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 parametrized 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 Jackson et al. [2001], the function *dom* that yields the domain of a relation is defined as

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

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:

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

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

Dynamic logic [Harel et al. 2000], 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

$$\begin{gathered}\alpha \\ \{A\} \\ \beta\end{gathered}$$

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

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

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 on the semantics, as well as the fact that actions can be sequentially composed, iterated or nondeterministically chosen, while Alloy functions cannot. Another relevant difference is that while functions in Alloy are shorthands for formulas that can be defined in RL, DynAlloy's actions are new elements in the language.

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

$$formula ::= \ldots \mid formula\ \{program\}\ formula \quad \text{“partial correctness”}$$

The syntax for programs is the one defined in Harel et al. [2000] for the class of regular programs plus a new rule to allow the construction of atomic actions from their pre and post conditions.

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

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

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

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

Fig. 7. Syntax and Semantics of DynAlloy.

In Figure 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

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

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 "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 $O \in \{Flush, SysWrite\}$ the following holds:

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

This proof method is sound but incomplete, since the invariance may be violated in unreachable 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 [Jackson et al. 2001].

The following example, extracted from Jackson et al. [2001], 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 unreachable states are no longer a burden because all 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 suitable. 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 action *SysWrite* is done as follows:

```
s = s0

  {SysWrite(s: System)}

some d: Data, a: Addr |
  s.cache = s0.cache ++ (a → d) ∧
  s.cache.dirty = s0.cache.dirty + a ∧
  s.main = s0.main
```

For action *Flush*, the specification becomes:

```
s = s0

  {Flush(s: System)}

some x: set s0.cache.addrs |
  s.cache.map = s0.cache.map - x→Data ∧
  s.cache.dirty = s0.cache.dirty - x  ∧
  s.main.map = s0.main.map ++ {a: x, d: Data | d = s0.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 article, 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 Jackson [2000], 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 [Dijkstra and Scholten 1990].

In the next paragraphs we will define a function

$$wlp : program \times formula \to formula$$

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: 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:

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

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 is equivalent 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 nondirty 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 the invariant DirtyInv. This property is specified in DynAlloy by:

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

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

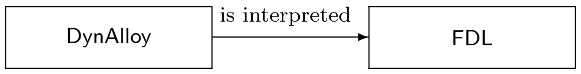

Fig. 8. Relationships among the formalisms DynAlloy and FDL.

specification is as follows:

$$\mathrm{s} = \mathrm{s}_0$$

$$\{\mathrm{DSFlush}(\mathrm{s} : \mathit{System})\}$$

$$\begin{aligned}&\mathrm{s.cache.dirty} = \emptyset \wedge\\ &\quad \mathrm{s.cache.map} = \mathrm{s}_0\mathrm{.cache.map} -\\ &\qquad \mathrm{s}_0\mathrm{.cache.map}[\mathrm{s}_0\mathrm{.cache.dirty}] \ \wedge\\ &\quad \mathrm{s.main.map} = \mathrm{s}_0\mathrm{.main.map} +\!+\ \mathrm{s}_0\mathrm{.cache.map}[\mathrm{s}_0\mathrm{.cache.dirty}]\end{aligned}$$

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

$$\mathrm{FullyAgree}(s : \mathit{System}) \iff s.\mathrm{cache.map}\ \mathit{in}\ s.\mathrm{main.map}.$$

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

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

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 Figure 1, in this Section we deal with the portion reproduced in Figure 8.

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

action ::= $a_1, \ldots a_k$ (atomic actions)
| *skip*
| action+action (non-deterministic 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)

Fig. 9. Syntax of dynamic logic.

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 Figure 9.

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 Figure 10.

6.2 FDL: Dynamic Logic over Fork Algebras

In Section 6.1 we introduced first-order dynamic logic. As with classical first-order logic, it is possible to extend the language of dynamic logic by adding new symbols, and also to add new axioms giving meaning to these. In order to define FDL, we include in the set of function symbols of signature Σ, the set of

$$
\begin{array}{l}
Q : \text{form} \to ST \to Boolean \\
P : \text{action} \to \mathcal{P}\,(ST \times ST) \\
Z : \text{expr} \to ST \to \mathbf{s} \\
\\
Q[p(t_1,\ldots,t_n)]\mu = (Z[t_1]\mu,\ldots,Z[t_n]\mu) \in env(p) \ \text{(atomic formula)} \\
Q[!F]\mu = \neg Q[F]\mu \\
Q[F \&\& G]\mu = Q[F]\mu \wedge Q[G]\mu \\
Q[F \mid\mid G]\mu = Q[F]\mu \vee Q[G]\mu \\
Q[all\ v : t \ \mid \ F]\mu = \bigwedge\{Q[F](\mu \oplus v \mapsto x) \mid x \in env(t)\} \\
Q[some\ v : t \ \mid \ F]\mu = \bigvee\{Q[F](\mu \oplus v \mapsto x) \mid x \in env(t)\} \\
Q[\ [a]F\ ]\mu = \bigwedge\{Q[F]\nu \mid \langle \mu, \nu \rangle \in P(a)\} \\
\\
P[a] = env(a) \ \text{(atomic action)} \\
P[skip] = \{\ \langle \mu, \mu \rangle : \mu \in ST\ \} \\
P[a+b] = P[a] \cup P[b] \\
P[a;b] = P[a] \circ P[b] \\
P[a^*] = (P[a])^* \\
P[\alpha?] = \{\ \langle \mu, \mu \rangle : Q[\alpha]\mu\ \} \\
\\
Z[v]\mu = \mu(v) \\
Z[f(t_1,\ldots,t_k)]\mu = env(f)(Z[t_1]\mu,\ldots,Z[t_k]\mu)
\end{array}
$$

Fig. 10. Semantics of dynamic logic.

constants $\{0, 1, Id\}$, the set of unary symbols $\{^-, \breve{}, ^*\}$, 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*. Once FDL is defined, the main result in this section (Theorem 6.4) is an interpretation of DynAlloy into FDL. That is, we will present a semantics–preserving mapping from DynAlloy formulas to FDL formulas. This will allow us, in Section 6.3, to present a complete calculus for FDL, which can be used for proving DynAlloy theorems.

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

We will call theories containing the identities (axioms) 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. Then $\mathcal{A}$ is a structure of the form $\langle A, +, \cdot, ^-, 0, 1, ;, \breve{}, Id, \nabla \rangle$ in which the semantics of action, function and predicate symbols is given through an environment *env*. Since Ψ is an FDL theory (and therefore satisfies the axioms for fork algebras), $\mathcal{A}$ is a fork algebra. Thus, by Frias et al. [1997] and Frias [2002, Theorem 4.2], $\mathfrak{A}$ is isomorphic to a proper fork algebra with domain B. Let $h : A \to B$ 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 makes use of Frias [2002, Theorem 4.2], heavily relies on the use of fork algebras rather than plain relation algebras [Tarski and Givant 1987]. A model for an 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 $\underline{\text{is}}$ a set of binary relations. This allows us to look at specifications in FRL, 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 an FDL environment $\widehat{e}$ such that*

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

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 Theorem 4.1.

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

Let α be atomic: $\alpha = t_1 int_2$.

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

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

$$\begin{aligned} & M[\beta\{p\}\gamma]e \\ = \ & M[\beta]e \Rightarrow \forall e_1 \left(\langle e, e_1\rangle \in P[p] \ \Rightarrow \ M[\gamma]e_1\right) \\ & \quad \{\text{by def. } M\} \\ = \ & N[\mathsf{RL} \mapsto \mathsf{FRL}(\beta)]e' \ \Rightarrow forall e'_1 \left(\langle e', e'_1\rangle \in P[p] \ \Rightarrow \ N[\mathsf{RL} \mapsto \mathsf{FRL}(\gamma)]e'_1\right) \\ & \quad \{\text{by Thm 4.2}\} \\ = \ & Q[\mathsf{RL} \mapsto \mathsf{FRL}(\beta)]e' \ \Rightarrow \forall e'_1 \left(\langle e', e'_1\rangle \in P[p] \ \Rightarrow \ Q[\mathsf{RL} \mapsto \mathsf{FRL}(\gamma)]e'_1\right) \\ & \quad \{\text{by Remark 1}\} \\ = \ & Q[\mathsf{RL} \mapsto \mathsf{FRL}(\beta) \ \Rightarrow \ [p]\mathsf{RL} \mapsto \mathsf{FRL}(\gamma)]\widehat{e} \\ & \quad \{\text{by semantics of FDL and definition of } \widehat{e}\} \\ = \ & Q[\mathsf{DynAlloy} \mapsto \mathsf{FDL}\,(\beta\{p\}\gamma)]\widehat{e}. \\ & \quad \{\text{by definition of DynAlloy} \mapsto \text{FDL}\} \end{aligned}$$

□

LEMMA 6.3. *Let α be a DynAlloy formula. Let $\widehat{e}$ be an FDL environment. Then, there exists 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 remains to define function A. Let a be an atomic action symbol. We define

$$A(a) = \{\langle e_1, e_2\rangle : \langle e'_1, e'_2\rangle \in \widehat{e}(a)\}.$$

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 DynAlloy $\mapsto$ FDL(α) is valid in FDL.*

PROOF.

$$\begin{aligned}
&\mathsf{DynAlloy} \mapsto \mathsf{FDL}(\alpha) \text{ is not valid in } \mathsf{FDL} & \\
\iff\ & \exists\widehat{e}\left(\neg Q[\mathsf{DynAlloy} \mapsto \mathsf{FDL}(\alpha)]\widehat{e}\right) & \text{(by semantics FDL)}\\
\iff\ & \exists e\left(\neg M[\alpha](e)\right) & \text{(by Lemmas 6.2 and 6.3)}\\
\iff\ & \alpha \text{ is not valid in } \mathsf{DynAlloy}. & \text{(by semantics DynAlloy)}
\end{aligned}$$

□

6.3 A Complete Calculus for FDL

In this section we present a complete calculus for FDL. Notice that due to Theorem 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 schemas 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 Harel et al. [2000, Theorem 15.1.4]. Joining this theorem with the completeness of the axiomatization of closure fork algebras [Frias 2002, Theorem 4.3], it follows that this calculus is complete with respect to the semantics of FDL.

7. VERIFYING ALLOY SPECIFICATIONS WITH PVS

As has been shown in previous sections, the extended kernel for Alloy that has been presented here is a language suitable for the description of systems

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

Fig. 11. Diagrammatic representation of sequents.

behavior. There are different options in order to reason about such descriptions. Techniques such as model checking, SAT solving and theorem proving give one the possibility of detecting system flaws during early stages of the design life cycle.

Regarding the problem of theorem proving, there are several theorem provers that can be used to carry out this task. Among them, we can mention Isabelle [Nipkow et al. 2002], HOL [Gordon and Melham 1993], Coq [Dowek et al. 1993] and PVS [Owre et al. 1992]. 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 [Owre et al. 1998]. Thus, we will concentrate on the use of this particular theorem prover in order to prove assertions from Alloy specifications.

As has been described in the basic PVS bibliography [Owre et al. 2001a, 2001b, 2001c], 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. Some distinguishing features of PVS for system specification are an advanced data-type specification language [Owre and Shankar 1993], the notion of subtypes and dependent types [Owre et al. 2001a], the possibility to define parametric theories [Owre et al. 2001a], and a collection of powerful proof commands to carry out propositional, equality, and arithmetic reasoning [Owre et al. 2001b]. 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 Figure 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 such as the one presented in Figure 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 Alloy specifications is not trivial because this language is not supported by the PVS tool itself. To bridge this gap, a proof checker was built by encoding the new semantics for Alloy into PVS' language [Lopez Pombo et al. 2002].

```
FODL_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 Formula_.''
    T?(f: Formula_): T??: Program_

``Constructs a (atomic action) Program_ from two Formula_.''
    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 FODL_Language
```

Fig. 12. Dynamic logic language encoded in as a PVS theory.

This framework is separated in two parts. The first one is a packet of thirteen files containing the theories needed to encode the fragment of the language that is common in every specification, for example, the file "FODL_Language.pvs", presented in Figure 12. This theory contains an encoding of the language of dynamic logic. Notice that the fact that this theory is common to every specification relies on the use of abstract data types, parametric theories, subtypes and dependent types.

The other files contain those theories that depend on the specification. Taking as a case study the memories with cache (systems) presented in Section 5.2, we provided five theories in order to build the PVS specification.

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

Fig. 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))))
```

Fig. 14. PVS translation of Formula (15).

Once the theories are built, we can start the theorem proving process. In Figures 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 Figures 13 and 14.

Notice that in the proof script shown in Figure 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:

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

In the proof script presented in Figure 15, these lemmas appear referenced by the names "SysWrite_preserves_DirtyInv" and "Flush_preserves_DirtyInv."

In the case of the proof script of Figure 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 various 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, Alloy formulas cannot be part of sequents unless they are conveniently preprocessed. This is why, given a formula α, we will prove the formula

```
FORALL (w : World_) : meaningF(f)(w).                (16)
```

rather than α itself. In Formula (16), function *meaningF* has type *Formula_* $\rightarrow$ *World_* $\rightarrow$ *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 Theorem 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 the user can simply declare

```
Theorem_1 : THEOREM f,
;;; 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))))))
```

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

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

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

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

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 unfolded the (implicit) occurrence of *meaningF* in the formula $(\alpha \vee \beta)(w)$, we would obtain the formula $(\alpha)(w) \vee (\beta)(w)$.

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

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

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:

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

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 to 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 skolemises a universal quantifier, and the star is used to tell PVS that it should skolemise 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 relies 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 partly because the use of lemmas simplifies the process of proving a property by allowing modular proofs. We recommend the reading of Lopez Pombo et al. [2002] as the reference material for this section.

The example presented is rather simple, but it shows that our extension of PVS can effectively be used for proving Alloy assertions. While there is no theoretical limitation to the Alloy models that can be translated and verified within PVS, modularization is the key toward successful theorem proving of Alloy assertions.

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 that 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 [Nipkow et al. 2002]. 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.

ACKNOWLEDGMENTS

We wish to thank Daniel Jackson for reading preliminary versions of this article and making valuable suggestions. Also, Sam Owre and Natarajan Shankar are to be thanked for their work on the verification of properties using PVS. Finally, the anonymous referees made criticisms that in our opinion led to a much better article than it was at first.

REFERENCES

ABRIAL, J. 1996. *The B-Book: assigning programs to meanings*. Cambridge University Press, New York, NY.

ARKOUDAS, K. 2000. Denotational proof languages. Ph.D. thesis, Massachusetts Institute of Technology.

ARKOUDAS, K., KHURSHID, S., MARINOV, D., AND RINARD, M. 2004. Integrating model checking and theorem proving for relational reasoning. In *Proceedings of the 7th Conference on Relational Methods in Computer Science (RelMiCS)–2nd. International Workshop on Applications of Kleene Algebra*, R. Berghammer and B. Möller, Eds. Lecture Notes in Computer Science, vol. 3051. Springer-Verlag, Malente, Germany, 204–213.

BICKFORD, M. AND GUASPARI, D. 1998. Lightweight analysis of UML. Tech. Rep. TM-98-0036, Odyssey Research Associates, Ithaca, NY, November.

BOOCH, G., RUMBAUGH, J., AND JACOBSON, I. 1998. *The Unified Modeling Language User Guide*. Addison–Wesley Longman Publishing Co., Inc., Boston, MA.

BURRIS, S. AND SANKAPPANAVAR, H. P. 1981. *A Course in Universal Algebra*. Graduate Texts in Mathematics. Springer-Verlag, Berlin, Germany.

CLARKE, E. M., GRUMBERG, O., AND PELED, D. 2000. *Model Checking*. MIT Press, Cambridge, MA.

CLEAVELAND, R., KLEIN, M., AND STEFFEN, B. 1993. Faster model checking for the modal mu-calculus. In *Proceedings of Computer Aided Verification (CAV)*, G. von Bochmann and D. K. Probst, Eds. Lecture Notes in Computer Science, vol. 663. Springer-Verlag, Montreal, Canada, 410–422.

DIJKSTRA, E. W. AND SCHOLTEN, C. S. 1990. *Predicate Calculus and Program Semantics*. Springer-Verlag, New York, NY.

DOWEK, G., FELTY, A., HERBELIN, H., HUET, G., MURTHY, C., PARENT, C., PAULIN-MOHRING, C., AND WERNER, B. 1993. The coq proof assistant user's guide (version 5.8). Tech. Rep. 154, INRIA, Rocquencourt, France.

FRIAS, M. F. 2002. *Fork Algebras in Algebra, Logic and Computer Science*. Advances in Logic, vol. 2. World Scientific Publishing Co., Singapore.

FRIAS, M. F., BAUM, G. A., AND MAIBAUM, T. S. E. 2002. Interpretability of first-order dynamic logic in a relational calculus. In *Proceedings of the 6th. Conference on Relational Methods in Computer Science (RelMiCS)—TARSKI*, H. de Swart, Ed. Lecture Notes in Computer Science, vol. 2561. Springer-Verlag, Oisterwijk, The Netherlands, 66–80.

FRIAS, M. F., HAEBERER, A. M., AND VELOSO, P. A. S. 1997. A finite axiomatization for fork algebras. *Logic Journal of the IGPL 5*, 3, 311–319.

Frias, M. F., Lopez Pombo, C. G., Baum, G. A., Aguirre, N. M., and Maibaum, T. S. E. 2003. Taking Alloy to the movies. In *Proceedings of FM 2003: the 12th International FME Symposium*, K. Araki, S. Gnesi, and D. Mandrioli, Eds. Lecture Notes in Computer Science, vol. 2805. Springer-Verlag, Pisa, Italy, 678–697.

Gordon, M. J. and Melham, T. F., Eds. 1993. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, New York, NY.

Harel, D., Kozen, D., and Tiuryn, J. 2000. *Dynamic Logic*. Foundations of Computing. MIT Press, Cambridge, MA.

Jackson, D. 2000. Automating first-order relational logic. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM Press, San Diego, California, 130–139.

Jackson, D. 2002a. Alloy: a lightweight object modelling notation. *ACM Trans. Soft. Eng. Meth. 11*, 2, 256–290.

Jackson, D. 2002b. *A Micromodel of Software: Lightweight Modelling and Analysis with Alloy*. MIT Laboratory for Computer Science, Cambridge, MA.

Jackson, D., Schechter, I., and Shlyahter, H. 2000. Alcoa: the alloy constraint analyzer. In *Proceedings of the 22nd. International Conference on Software Engineering*, C. Ghezzi, M. Jazayeri, and A. L. Wolf, Eds. Association for the Computer Machinery and IEEE Computer Society, ACM Press, Limerick, Ireland, 730–733.

Jackson, D., Shlyakhter, I., and Sridharan, M. 2001. A micromodularity mechanism. In *Proceedings of the 8th European Software Engineering Conference* held together with the *9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM Press, Vienna, Austria, 62–73.

Jackson, D. and Sullivan, K. 2000. COM revisited: tool-assisted modelling of an architectural framework. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-First Century Applications*, D. S. Rosenblum, Ed. ACM Press, San Diego, CA, 149–158.

Jones, C. 1986. *Systematic Software Development Using VDM*. Prentice Hall, Hertfordshire, UK.

Lopez Pombo, C. G., Owre, S., and Shankar, N. 2002. A semantic embedding of the $\mathbf{A_g}$ dynamic logic in PVS. Tech. Rep. SRI-CSL-02-04, Computer Science Laboratory, SRI International. July.

Manna, Z., Anuchitanukul, A., Bjørner, N., Browne, A., Chang, E., Colon, M., de Alfaro, L., Devarajan, H., Sipma, H., and Uribe, T. 1994. STeP: The stanford temporal prover. Tech. Rep., Stanford University, Stanford, CA.

McMillan, K. L. 1993. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA.

Nipkow, T., Paulson, L. C., and Wenzel, M. 2002. *Isabelle/HOL—A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, vol. 2283. Springer-Verlag, Berlin, Germany.

Object Management Group. 1997. *Object Constraint Language Specification*. Object Management Group, Needham, MA, version 1.1.

Owre, S., Rushby, J. M., and Shankar, N. 1992. PVS: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction (CADE)*, D. Kapur, Ed. Lecture Notes in Artificial Intelligence, vol. 607. Springer-Verlag, Saratoga, NY, 148–752.

Owre, S., Rushby, J. M., Shankar, N., and Stringer-Calvert, D. 1998. PVS: an experience report. In *Proceedings of Applied Formal Methods – (FM-Trends) '98*, D. Hutter, W. Stephan, P. Traverso, and M. Ullman, Eds. Lecture Notes in Computer Science, vol. 1641. Springer-Verlag, Boppard, Germany, 338–345.

Owre, S. and Shankar, N. 1993. Abstract datatypes in PVS. Tech. Rep. SRI-CSL-93-9R, Computer Science Laboratory, SRI International. December. Subtantially revised in June 1997.

Owre, S., Shankar, N., Rushby, J. M., and Stringer-Calvert, D. 2001a. *PVS Language Reference*, Version 2.4 ed. SRI International.

Owre, S., Shankar, N., Rushby, J. M., and Stringer-Calvert, D. 2001b. *PVS Prover Guide*, Version 2.4 ed. Computer Science Laboratory, SRI International.

Owre, S., Shankar, N., Rushby, J. M., and Stringer-Calvert, D. 2001c. *PVS System Guide*, Version 2.4 ed. Computer Science Laboratory, SRI International.

SPIVEY, J. M. 1988. *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge University Press, New York, NY.

TARSKI, A. AND GIVANT, S. 1987. *A Formalization of Set Theory Without Variables*. American Mathematical Society Colloqium Publications, Providence, RI.

VARDI, M. Y. AND WOLPER, P. 1986. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings of the Symposium on Logic in Computer Science '86*, A. Meyer, Ed. IEEE Computer Society, Cambridge, MA, 332–344.

Received April 2003; revised August 2004 and April 2005; accepted June 2005