Alloy Analyzer+PVS in the Analysis and Verification of Alloy Specifications

Marcelo F. Frias[⋆], Carlos G. Lopez Pombo, and Mariano M. Moscato

Department of Computer Science, FCEyN,
Universidad de Buenos Aires and CONICET.
e-mail: {mfrias, clpombo, mmoscato}@dc.uba.ar

Abstract. This article contains two main contributions. On the theoretical side, it presents a novel complete proof calculus for Alloy. On the applied side we present Dynamite, a tool that combines the semi-automatic theorem prover PVS with the Alloy Analyzer. Dynamite allows one to prove an Alloy assertion from an Alloy specification using PVS, while using the Alloy Analyzer for the automated analysis of hypotheses introduced during the proof process. As a means to assess the usability of the tool, we present a complex case-study based on Zave's Alloy model of addressing for interoperating networks.

1 Introduction

Alloy [6] is a formal modeling language with a simple syntax based on notations ubiquitous in object orientation, and semantics based on relations. Part of its appeal comes from the existence of the Alloy Analyzer, which allows one to analyze Alloy specifications in a fully automatic way. The analysis process relies on a translation of Alloy specifications (where domains are bounded to finite sizes) to a propositional formula, which is then analyzed using off-the-shelf SAT-solvers. Bounding the size of domains has a direct impact on the conclusions we can draw from the analysis process. If a counterexample for a given assertion is found, then the model is for sure flawed. On the other hand, if no counterexample is found, we can only conclude that no counterexamples exist when domain sizes are constrained to the given bounds. Choosing larger bounds may show the existence of previously unforeseen errors. This limited analyzability offered by the Alloy Analyzer is essential in order to analyze Alloy models and get rid of most errors introduced in the modeling process. At the same time, models for critical applications can also benefit from usage of the Alloy Analyzer, but one cannot entirely rely on that.

Lightweight formal methods with the limitations of the Alloy Analyzer cannot be entirely trusted when dealing with critical models. An alternative is the use of heavyweight formal methods, as for instance semi-automatic theorem provers, and among these, PVS [9]. Theorem provers have limitations too. First, they

[⋆] A preliminary version of this paper will be presented at the First Alloy Workshop, colocated with 14th ACM Symposium on Foundations of Software Engineering, 2006.

require an expertise from the user that many times discourages their use. Also, theorem provers use their own languages which are seldom close to lightweight modeling languages. This detracts from their usability by lightweight users, and is also a source of errors in case lightweight models have to be translated. Equally important, minor errors in a model may require to redo proofs that were using wrong hypotheses. Much the same as errors overlooked during software requirement elicitation have a greater impact the more advanced the development stage, model errors have greater impact the more auxiliary lemmas have been proved. Therefore, getting rid of as many errors as possible from the model before starting the theorem proving process is a must.

The previous paragraphs show that a marriage between simple automatically analyzable formal modeling languages and semiautomatic theorem provers is in fact necessary when analyzing critical models. The goal of this paper is to present Dynamite, a tool that marries Alloy and PVS.

Dynamite is an extension of PVS that incorporates the following features:

1. Sound automatic translation of Alloy models to PVS, preventing the introduction of errors in the translation process.
2. Novel complete proof calculus for Alloy. Therefore, valid Alloy properties can be proved (although this requires interaction with the user).
3. Modified PVS pretty-printer that shows proof steps in Alloy language (thus bridging the gap between Alloy and PVS).
4. Fluid automatic interaction with the Alloy Analyzer in order to automatically analyze hypotheses introduced during the theorem proving process.

The contributions of this paper are the following:

1. We present a novel complete calculus for Alloy by interpreting Alloy theories to fork algebra [3] theories.
2. We present Dynamite, the tool that incorporates the previously enumerated features.
3. We give a brief description of a case study where we prove several assertions introduced in Zave's Alloy model of addressing for interoperating networks [11], and present some conclusions regarding the usability and limitations of Dynamite.

The article is organized as follows. In Section 2 we present the Alloy modeling language by means of an example, as well as its supporting tool, the Alloy Analyzer. In Section 3 we present the complete calculus for Alloy. In Section 4 we describe our tool, Dynamite. In Section 5 we discuss a complex case-study. Finally, in Section 6 we discuss related work, conclusions about the contributions of this article, and some proposals for further work.

2 Alloy and the Alloy Analyzer

In this section we introduce the Alloy modeling language by means of an example. In [11] Zave presents a formal model of addressing for interoperating networks. These networks connect *agents* (which might be hardware devices or other software systems). Agents can be divided between *client agents* (users of the networking infrastructure), or *server agents* (part of the infrastructure). Agents can use resources from domains, to which they must be *attached.* In order to be able to reach clients from domains, pairs $\langle address, domain \rangle$ are assigned to clients. Different sorts of objects can be distinguished in the previous description. Signatures (akin to classes in object orientation), are the means to declare object domains. Figure 1 presents a (simplfied[1]) description of the signatures for this model.

```
sig Address{ }
sig Agent{ attachments: set Domain }
sig Server extends Agent { }
sig Client extends Agent { knownAt: Address -> Domain }
sig Domain{ space: set Address, map: space -> Agent }
```

Fig. 1. Simplified model for addresses, agents and domains.

Signature `Address` denotes a unary relation (set) whose objects are atomic. According to Alloy's formal semantics, signature `Agent` declares a set of objects *Agent*, and field `attachments` denotes a binary relation $attachments \subseteq Agent \times Domain$ (where *Domain* is the set denoted by signature `Domain`). Notice that without the modifier `set` in the declaration of field `attachments`, relation *attachments* would instead be a total function. Signature extension allows us to model single inheritance between signatures. Signature `Server` singles out some agents as *servers*. Signature `Client`, besides distinguishing some agents as *clients*, introduces a new field. Field `knownAt` allows us to retrieve the pairs $\langle address, domain \rangle$ mentioned above. Following Alloy's semantics, field `knownAt` denotes a *ternary* relation $knownAt \subseteq Client \times Address \times Domain$.

Axioms are included in a model under the form of *facts*. Recalling that dot (.) stands for composition of relations (called *navigation* in Alloy), an axiom saying that "whenever an agent appears in the range of the `map` attribute it is because the agent is attached to that domain", is written in Alloy as:

```
fact { all d: Domain, g: Agent |
             g in Address.(d.map) => d in g.attachments }
```

Besides navigation, the relational logic underlying Alloy [6] includes operations for union of relations (+), intersection (&), difference ($-$), transposition (which flips pairs $\langle x, y \rangle$ of a binary relation to $\langle y, x \rangle$ and is denoted by $\sim$) and reflexive-transitive closure ($*$). Also, *iden* denotes the binary identity relation, and *univ* denotes the unary universal relation (the set holding the union of all the domains from the model).

[1] The complete Alloy model can be obtained from `http://www.research.att.com/~pamela/svcrte.html`.

Once a model has been provided, it can be analyzed by looking for counterexamples of properties that are expected to hold in the model. These properties are called *assertions*. For instance, the (not necessarily valid) assertion that the `map` field targets only client agents, is written in Alloy as:

```
assert mapTargetsClients { all d:Domain, s:d.space |
                                        s.(d.map) in Client }
```

Command `check mapTargetsClients for 5` allows one to search for counterexamples in which domains have up to 5 elements, using the Alloy Analyzer. The Alloy Analyzer translates the model and the negation of the assertion to a propositional formula. Of course, the translation heavily depends on the bounds declared in the `check` command. Once a propositional formula has been obtained, the Alloy Analyzer employs off-the-shelf SAT-solvers, and in case a model of the formula is obtained, it is translated back to a counterexample of the source model and presented to the user using different visualization algorithms.

3 A Novel Complete Calculus for Alloy

Among the extensions of PVS included in Dynamite, an essential one is the inclusion of a complete calculus for Alloy. The calculus is obtained by translating Alloy theories (specifications) to fork algebraic theories (to be introduced in Section 3.1). Since:

1. the translation is semantics-preserving, and
2. there is a complete calculus for the fork algebras we will use in this article,

we will prove a theorem stating that an assertion α (semantically) follows from an Alloy specification Σ if (and only if) its translation $T(\alpha)$ can be proved from the translation of the theory using the complete calculus for fork algebras. In symbols, $\Sigma \models \alpha \iff \{ T(\sigma) : \sigma \in \Sigma \} \vdash T(\alpha)$. This kind of theorems relating two logics are often called interpretation theorems.

The following question is now likely to arise:

> *Is the fork algebra language substantially different from the Alloy language (therefore reducing the usability of the proposed calculus by current Alloy users)?*

This section is then structured as follows. In Section 3.1 we introduce the class of point-dense omega closure fork algebras (also noted as PDOCFA), as well as their complete calculus. In Section 3.2 we present the translation from Alloy formulas to formulas in the language of PDOCFA, and provide a proof sketch of the interpretation theorem. Since we aim at defining a translation that yields algebraic formulas as close to Alloy formulas as possible, in Section 3.3 we give an answer to the previous question and analyze the similarities and differences between the Alloy language and the fork algebraic language.

3.1 Point-Dense Omega Closure Fork Algebras

Proper Point-Dense Omega Closure Fork Algebras These algebras are structures whose domain is a set of binary relations built on top of a base set B. If we call R the domain of such an algebra (notice that $R \subseteq Pw\,(B \times B)$), R has to be closed under the following operations for sets: *union* $(+)$, *intersection* $(\&)$, *complement* (denoted, for a binary relation r, by $\overline{r}$), the *empty* binary relation $(\emptyset)$, and the *universal* binary relation, (usually $B \times B$, and denoted by 1).

Besides the previous operations for sets, R has to be closed under the following operations for binary relations: *transposition* $(\sim)$, *navigation* $(.)$, and *reflexive–transitive closure* $(*)$. The *identity* relation (on B), is denoted by *iden*.

A binary operation called *fork* (∇) is included, which requires set B to be closed under an injective function $\star$. This means that there are elements x in B that are the result of applying the function $\star$ to elements y and z (i.e., $x = y \star z$). Since $\star$ is injective, x can be seen as an encoding of the pair $\langle y, z \rangle$. Those elements that do not encode pairs, are called *urelements*. Operation fork is defined by:

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

Finally, we require set R to be *point-dense*. A point is a relation of the form $\{\, \langle a, a \rangle \,\}$. Point-density requires set R to have plenty of these relations. More formally speaking, for each nonempty relation I contained in the identity relation there must be a point $p \in R$ satisfying $p \subseteq I$.

A Complete Calculus for Point-Dense Omega Closure Fork Agebras Introducing a calculus requires presenting its axioms and inference rules. Before doing so, we introduce some notation. In a proper PDOCFA the relations π and ρ defined by $\pi = \sim (iden \nabla 1)$ and $\rho = \sim (1 \nabla iden)$ behave as projections with respect to the encoding of pairs induced by the injective function $\star$. Their semantics in a proper PDOCFA $\mathfrak{A}$ whose binary relations range over a set B, is given by $\pi = \{\, \langle a \star b, a \rangle : a, b \in B \,\}$, $\rho = \{\, \langle a \star b, b \rangle : a, b \in B \,\}$.

The operation *cross* $(\otimes)$ performs a parallel product. Its set-theoretical definition is given by $r \otimes s = \{\, \langle a \star c, b \star d \rangle : \langle a, b \rangle \in r \text{ and } \langle c, d \rangle \in s \,\}$. In algebraic terms, operation cross is defined by $r \otimes s = (\pi . r) \,\nabla\, (\rho . s)$.

We can characterize points as nonempty relations that satisfy the property $x.1.x \subseteq iden$. If we denote the inclusion relation by "in" (as in Alloy), the predicate "Point" defined by "Point$(p) \iff p \mathrel{!=} \emptyset \;\&\&\; p.1.p$ in $iden$" determines those relations that are points. The axioms and inference rules for the calculus are then:

1. Axioms for Boolean algebras defining the meaning of $+$, $\&$, $^{-}$, $\emptyset$ and 1.
2. Formulas defining composition of binary relations, transposition, reflexive–transitive closure and the identity relation:
   $x.(y.z) = (x.y).z$,
   $x.iden = iden.x = x$,
   $(x.y)\,\&z = \emptyset$ iff $(z.\sim y)\,\&x = \emptyset$ iff $(\sim x.z)\,\&y = \emptyset$,
   $*x = iden + (x.*x)$,
   $*x.y.1$ in $(y.1) + \left(*x.(\overline{y.1}\ \&\ (x.y.1))\right)$.

3. Formulas defining the operator ∇:
   $x \nabla y = (x . \sim \pi) \,\&\, (y . \sim \rho)$,
   $(x \nabla y) . \sim (w \nabla z) = (x . \sim w) \,\&\, (y . \sim z)$,
   $\pi \nabla \rho$ in *iden*.
4. A formula enforcing point-density:
   all x | $(x$!= $\emptyset$ && x in *iden*$)$ => (some p | Point(p) && p in $x)$,
5. Term $\overline{1 . (1 \nabla 1)}$ & *iden* (to be abbreviated as $iden_{\mathsf{U}}$) defines a partial identity on the set of urelements. Then, the following formula forces the existence of a nonempty set of urelements:
   $1 . iden_{\mathsf{U}} . 1 = 1$

The inference rules for the closure fork calculus are those for classical first-order logic (choose you favorite ones), plus the following equational (but infinitary) proof rule for reflexive-transitive closure[2]:

$$\frac{\vdash iden \text{ in } y \quad x^i \text{ in } y \vdash x^{i+1} \text{ in } y}{\vdash *x \text{ in } y} \;\; (\mathit{\Omega\ Rule})$$

The axioms and rules given above define a class of models. Proper PDOCFA satisfy the axioms [4], and therefore belong to this class. It could be the case that there are models for the axioms that are not proper PDOCFA. Fortunately, the following theorem (which follows from [4], [3, Thm. 4.2], [8, Thm. 52]), states that if a model is not a proper PDOCFA then it is isomorphic to one.

Theorem 1. *Every* PDOCFA $\mathfrak{A}$ *is isomorphic to a proper* PDOCFA $\mathfrak{B}$. *Moreover, there exist relations* $\{ \langle a_0, a_0 \rangle \}, \ldots, \{ \langle a_i, a_i \rangle \} \ldots$ *(possibly infinitely many of them) that belong to* $\mathfrak{B}$*, such that iden* $= \{ \langle a_0, a_0 \rangle, \ldots, \langle a_i, a_i \rangle, \ldots \}$.

Constraining Quantifiers to Atoms Alloy quantifiers range over relations of the form $\{ a \}$, i.e., over unary singletons. On the other hand, relational quantifiers range over the elements of a PDOCFA, which are not even required to be relations (recall that PDOCFAs are just models of a set of axioms). But, since PDOCFAs are all isomorphic to proper ones, a relational quantifier can always be seen as ranging over all binary relations from a proper PDOCFA. Still a big distance remains between unary singletons and arbitrary binary relations. It is at least obvious that there are many more of the latter, than there are of the former. Point-density, by forcing the existence of all singletons, allows us to establish a one-one correspondence between $\{ a \}$ and $\{ \langle a, a \rangle \}$. Therefore, we will mimic the behavior of Alloy quantifiers by constraining relational quantifiers to range over points. Notice that some points hold urelements, but others do not. In this case, since Alloy atoms do not have structure (the structure is modeled through fields), we will employ points holding urelements.

We will now consider the restricted part of the first-order language of PDOCFAs defined by the following grammar:

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

F ::= Equation | !F | F1 || F2 | F1 && F2 |
 ::= all p / (Point(p) && p in $iden_{\mathsf{U}}$) implies F

Actually, in a PDOCFA we will have different sub relations of $iden_{\mathsf{U}}$, namely $iden_1, \ldots, iden_k$, representing each one a different Alloy signature $\mathrm{sig}_1, \ldots, \mathrm{sig}_k$. We will then use the following abbreviated notation for formulas. The formula "all p | (Point(p) && p in $iden_i$) implies F" is denoted as "all p : sig_i | F". Similar abbreviations are used for the "some" quantifier.

3.2 A Complete Calculus for Alloy

In this section we introduce a mapping from Alloy formulas to formulas in the language defined in Section 3.1. The mapping keeps the structure of Alloy formulas almost unchanged, thus simplifying the understanding of the resulting formulas by casual Alloy users. Since PDOCFAs only contain binary relations, we will show how to model relations of arbitrary rank as binary ones, with the aid of fork. We then prove that the resulting calculus is complete for Alloy.

Handling Relations of Rank Greater Than Two Recall that due to the fork operator, the underlying domain of a proper PDOCFA is closed under an injective operation $\star$. Given a n-ary relation $R \subseteq A_1 \times \cdots \times A_n$, we will represent it by the binary relation

$$\{\, \langle a_1, a_2 \star \cdots \star a_n \rangle : \langle a_1, \ldots, a_n \rangle \in R \,\} \ .$$

This will be an invariant in the representation of n-ary relations by binary ones. For instance, ternary relation `knownAt` is encoded as a binary relation $\mathsf{knownAt}$ whose elements are pairs of the form $\langle c, a \star d \rangle$ for $c : \mathit{Client}$, $a : \mathit{Address}$ and $d : \mathit{Domain}$. We will in general denote the encoding of a relation C as a binary relation, by C. Given a point $c : \mathit{Client}$, the navigation of the relation $\mathsf{knownAt}$ through c should result in a binary relation contained in $\mathit{Address} \times \mathit{Domain}$. Given a point $a : t$ and a binary relation R encoding a relation of rank higher than 2, we define the navigation operation $\bullet$ by

$$a \bullet R = \sim \pi \,.\, \mathit{Ran}\,(a.R) \,.\, \rho \ . \tag{1}$$

Operation Ran in (1) returns the range of a relation as a partial identity. It is defined by $\mathit{Ran}\,(x) = (x.1)\,\&\mathit{iden}$. Its semantics in terms of binary relations is given by $\mathit{Ran}\,(R) = \{\, \langle a, a \rangle : \text{some } b \mid \langle b, a \rangle \in R \,\}$.

For a binary relation R representing a relation of rank less than or equal to 2, navigation is easier. Given a point $a : t$, we define

$$a \bullet R = \mathit{Ran}\,(a.R) \ .$$

It still remains to define navigation whenever the relation on the left-hand side is not a point, i.e., it has rank greater than 1. The definition is as follows:

$$R \bullet S = \begin{cases} R.\,(\mathit{iden} \otimes (\mathit{iden} \otimes (\cdots \otimes ((\mathit{iden} \otimes S)\,.\pi)))) & \text{if } \mathrm{rank}(S) = 1 \\ R.\,(\mathit{iden} \otimes (\mathit{iden} \otimes (\cdots \otimes (\mathit{iden} \otimes S)))) & \text{if } \mathrm{rank}(S) > 1 \end{cases}$$

Going back to our example about agents, it is easy to check that for a point $c' : \mathit{Client}$ such that $c' = \{ \langle c, c \rangle \}$,

$$c' \bullet \mathsf{knownAt} = \{ \langle a, d \rangle : a \in \mathit{Address}, d \in \mathit{Domain} \ \mathit{and} \ \langle c, a \star d \rangle \in \mathsf{knownAt} \} \ .$$

Translating Alloy Formulas to Relational Formulas In this section we present a translation of Alloy formulas to formulas in the language of PDOCFAs. Prior to that, it is necessary to translate Alloy terms to fork-algebra terms.

$$\begin{array}{llll} T(C) & = \mathsf{C}, & T(r{+}s) & = T(r) + T(s), \\ T(x_i) & = X_i, (X_i \text{ variable ranging over points}) & T(r\&s) & = T(r)\&T(s), \\ T(\sim r) & = \sim T(r), & T(r-s) & = T(r)\&\overline{T(s)}, \\ T(*r) & = *T(r), & T(r.s) & = T(r) \bullet T(s) \end{array}$$

We are now in the right conditions for translating formulas. The translation differs from the one previously presented in [5] in that the target of the translation is a first-order language rather than an equational language, and therefore it is no longer necessary to encode quantified variables because these are kept explicit. This will greatly improve the understandability of the translation by a casual Alloy user.

$$\begin{array}{llll} F(t_1 \text{ in } t_2) & = T(t_1) \text{ in } T(t_2), & F(\alpha \ \&\& \ \beta) & = F(\alpha) \ \&\& \ F(\beta), \\ F(!\alpha) & = !F(\alpha), & F(\text{all } x : S \mid \alpha) & = \text{all } x : S \mid F(\alpha), \\ F(\alpha \ || \ \beta) & = F(\alpha) \ || \ F(\beta), & F(\text{some } x : S \mid \alpha) & = \text{some } x : S \mid F(\alpha). \end{array}$$

Recall that quantifications in the right-hand side are abbreviations for formulas where quantifiers range over points of the appropriate signature. Notice that formulas are undistinguishable from Alloy formulas.

Completeness of the Alloy Calculus Formal semantics of Alloy assigns semantics to expressions and formulas in a given *environment*. An environment is a function that assigns sets to signatures, adequate relations to relational constants (those arising from signature fields), and values to variables over individuals. From an Alloy environment e we build a PDOCFA $\mathfrak{F}_e$ and a relational environment e' as follows:

– Let $\text{sig}_1, \ldots, \text{sig}_k$ be the Alloy signatures. Let $A = \bigcup_{1 \leq i \leq k} e\,(\text{sig}_i)$. Let $\mathfrak{T}(A)$ be the set of finite binary trees with information in the leaves, and whose information are elements from A.
– Let $\mathfrak{F}_e$ be the omega closure fork algebra with universe $\mathit{Pw}\,(\mathfrak{T}(A) \times \mathfrak{T}(A))$. If we denote the tree constructors by: $\mathsf{leaf} : A \to \mathfrak{T}(A)$ and $\mathsf{bin} : \mathfrak{T}(A) \times \mathfrak{T}(A) \to \mathfrak{T}(A)$, the fork operation is defined by

$$R \nabla S = \{ \langle a, \mathsf{bin}(b, c) \rangle : \langle a, b \rangle \in R \ \wedge \ \langle a, c \rangle \in S \} \ .$$

  Notice that the remaining operations have their meaning fixed once the domain $\mathit{Pw}\,(\mathfrak{T}(A) \times \mathfrak{T}(A))$ is fixed.

– Let e' be the environment satisfying:
  - $e'(\mathrm{sig}_i) = \left\{ p \in \mathfrak{F}_e : \mathrm{Point}(p) \wedge p \leq \mathit{iden}_{e(\mathrm{sig}_i)} \right\}$,
  - $e'(R) = \mathsf{R}$ (the binary encoding of relation $e(R)$),
  - $e'(v_i) = \{ \langle e(v_i), e(v_i) \rangle \}$.

Similarly, given a proper PDOCFA, and a relational environment e, we define an Alloy environment e' as follows:

- $e'(\mathrm{sig}_i) = \left\{ a : \langle a, a \rangle \in \mathit{iden}_{\mathrm{sig}_i} \right\}$,
- $e'(R) = \{ \langle a_1, \ldots, a_n \rangle : \langle a_1, a_2 \star \cdots \star a_n \rangle \in e(\mathsf{R}) \}$,
- $e'(v_i) = a$ such that $e(v_i) = \{ \langle a, a \rangle \}$.

From the previous definitions, the following lemmata can be proved by induction on the structure of Alloy formulas. The proofs are not included due to the lack of space, but follow the lines of previous interpretability results by the authors [3, 5].

Lemma 1. *Given an Alloy environment e, $\models \varphi[e] \iff \mathfrak{F}_e \models F(\varphi)[e']$.*

Lemma 2. *Given a PDOCFA $\mathfrak{F}$ and a relational environment e, there exists an Alloy environment e' such that for every Alloy formula φ, $\mathfrak{F} \models F(\varphi)[e] \iff \models \varphi[e']$.*

We then prove the following completeness theorem. The turnstile symbol $\vdash$ stands for derivability in the calculus for PDOCFAs.

Theorem 2. *Let $\Sigma \cup \{ \varphi \}$ be a set of Alloy formulas. Then,*

$$\Sigma \models \varphi \iff \{ F(\sigma) : \sigma \in \Sigma \} \vdash F(\varphi).$$

Proof. $\Longrightarrow$) If $\{ F(\sigma) : \sigma \in \Sigma \} \nvdash F(\varphi)$, then there exists a PDOCFA $\mathfrak{F}$ such that $\mathfrak{F} \models \{ F(\sigma) : \sigma \in \Sigma \}$ and $\mathfrak{F} \not\models F(\varphi)$. From Thm. 1 there exists a proper PDOCFA $\mathfrak{F}'$ isomorphic to $\mathfrak{F}$. Clearly, $\mathfrak{F}' \models \{ F(\sigma) : \sigma \in \Sigma \}$ and $\mathfrak{F}' \not\models F(\varphi)$. Then, there is a relational environment e such that $\mathfrak{F}' \models \{ F(\sigma) : \sigma \in \Sigma \} [e]$ and $\mathfrak{F}' \not\models F(\varphi)[e]$. From Lemma 2, there exists an Alloy environment e' such that $\models \Sigma[e']$ and $\not\models \varphi[e']$. Thus, $\Sigma \not\models \varphi$.
$\Longleftarrow$) If $\Sigma \not\models \varphi$, then there exists an Alloy environment e such that $\models \Sigma[e]$ and $\not\models \varphi[e]$. From Lemma 1 there exist a proper PDOCFA $\mathfrak{F}_e$ and a relational environment e' such that $\mathfrak{F}_e \models \{ F(\sigma) : \sigma \in \Sigma \} [e']$ and $\mathfrak{F}_e \not\models F(\varphi)[e']$. Then, $\{ F(\sigma) : \sigma \in \Sigma \} \nvdash F(\varphi)$.

3.3 Comparing the Source and Target Formalisms

If the calculus introduced in Section 3.2 is to be used by Alloy users, then the language should be as close as possible to Alloy. The translation of formulas shows that the formulas result of applying the translation (we are not discussing terms yet) are indeed Alloy formulas. It is clear that Alloy operations have a direct algebraic counterpart. Thus, from a syntactical point of view, terms result of the translation are also Alloy terms. There are a few points that need to be addressed, though. Namely:

1. Atoms (which in Alloy are modeled as unary singletons $\{ a \}$) are modeled in the algebraic setting as singleton binary relations $\{ \langle a, a \rangle \}$.
2. More generally, relations that may have rank greater than 2 in Alloy, are modeled in the algebraic setting as binary relations.

In our experience it is seldom the case that two relations having rank greater than 2 are composed. The most common situation arises when an atom is composed with a relation of higher rank ($a.R$). We provide in Dynamite a theory for fork algebras that includes proofs of the usual properties of composition, as for instance

$$\text{all } R, S, T \mid R \text{ in } S \text{ implies } (R \bullet T \text{ in } S \bullet T) \ \&\& \ (T \bullet R \text{ in } T \bullet S) \quad \text{monotonicity}$$
$$\text{all } R, S, T \mid !\text{Set}(S) \text{ implies } (R \bullet S) \bullet T = R \bullet (S \bullet T) \quad \text{associativity}$$

Proving these properties requires using the full power of the calculus, including quantifications over relations, which cannot even be expressed in Alloy. These are part of the infrastructure provided by Dynamite. A user can prove particular instances of (for example) monotonicity with respect to fields F_1, F_2, F_3 (provided by the Alloy model) by instantiating the previous properties. She can also prove the property from scratch for the particular instances using Dynamite.

4 The Dynamite Tool

PVS [9] interacts with its users through the highly customizable text editor EMACS. Dynamite is a tool developed by customizing both EMACS and PVS. In Sections 4.1 and 4.2 we describe these customizations. In Section 4.3 we describe the proof process a user would go through, showing how these adaptations make the proof process more amenable.

4.1 Customizations Made on EMACS

EMACS is a highly customizable text editor. It is possible to run other applications from within EMACS. It is now possible to run the Alloy Analyzer on a specific model in order to analyze a provided assertion. While the standard scope for domains is 3, it is also possible for the user to choose new scopes. This is extremely useful when adding lemmas whose proof has not yet been developed, to a theory. The new lemma can be checked within the theory both for counterexamples and consistency with the aid of the Alloy Analyzer. Once PVS has been started, it is possible to choose an Alloy model (a .als file) and an extension of EMACS allows one to translate the Alloy model to an appropriate PVS theory.

4.2 Customizations Made on PVS

PVS reads theories and shows proofs in its specific syntax. Even properties written in Alloy, if one wants to prove them with the support of PVS, have to be rewritten using the syntax PVS recognizes. We have modified the PVS pretty printer in order to exhibit formulas using Alloy syntax. This will be shown with an example in Section 5.

The PVS rule "case", which allows one to introduce new hypotheses along a proof, has also been modified. According to [9], if the current sequent is of the form $\Gamma \vdash \Delta$, then the rule "(case A)" generates the subgoals $A, \Gamma \vdash \Delta$ and $\Gamma \vdash A, \Delta$. The rule allows to use formula A as an extra hypothesis along the proof of Δ, which has to be discharged later through a proof. Executing the modified rule "case", besides performing its regular duty of generating the appropriate subgoals, also automatically analyzes formula A using the Alloy Analyzer.

4.3 A Proof Scenario

A development team has built an Alloy model for a critical domain, and has already debugged it by automatically analyzing (using the Alloy Analyzer) some appropriate assertions. Since the model will serve as a basis for the development of a critical system, bounded analysis is not enough. The team then faces the need to prove a given property about their model. Upon starting Dynamite, they choose to upload the Alloy model. This generates (although they do not need to know about it), the corresponding PVS theory, and the user can choose an assertion to prove. Facts from the model are now available as axioms to be used in proofs.

The proof then proceeds until a new hypothesis has to be introduced using the PVS command "case", in whose case the Alloy Analyzer is lunched in the background in order to check the hypothesis for counterexamples and consistency. If a new lemma has to be added to the theory, then the Alloy Analyzer can be used from within the framework in order to check for the existence of counterexamples and for consistency, too.

5 A Case Study: A Formal Model of Addressing for Interoperating Networks

In her paper [11], Zave presents a formal model of addressing for interoperating networks. Part of the model is presented in Fig. 1. Domains can create persistent connections between agents. Such connections are called *hops*. Besides the domain that created it, a hop contains information about the initiator and acceptor agents taking part in the connection, and also source and target addresses. A fact forces these addresses to correspond to the agents (according to the domain map).

```
sig Hop{ domain: Domain,
         initiator, acceptor: Agent,
         source, target: Address }
```

Multi-hop connections are enabled by the servers. These connections are called *links*. Links contain information about the server enabling the connection, and about the connected hops.

```
abstract sig End { }
one sig Init, Accept extends End { }
sig Link{ agent:Server, oneHop,anotherHop:Hop, oneEnd,anotherEnd:End }
{  oneHop != anotherHop
   oneEnd in Init => agent=oneHop.initiator
   oneEnd in Accept => agent=oneHop.acceptor
   anotherEnd in Init => agent=anotherHop.initiator
   anotherEnd in Accept => agent=anotherHop.acceptor }
```

The reflexive-transitive closure of the accessibility relation determined by links is kept by an object "Connections", which also keeps the relation established by the links.

```
one sig Connections{ atomConnected, connected: Hop -> Hop }
```

Interoperation is considered a *feature* of networks. Features are installed in domains and have a set of servers from that domain that implement them. Among the facts related to features, we find that each feature has at least one server, and that each server implements exactly one feature.

```
abstract sig Feature { domain: Domain, servers: set Server }
```

Interoperation features are then characterized as follows:

```
sig InteropFeature extends Feature{
   toDomain: Domain,
   exported, imported, remote, local: set Address,
   interTrans: exported some -> some imported }
{  domain != toDomain
   exported in domain.space   && remote in exported
   imported in toDomain.space && local in imported
   remote.interTrans = local }
```

An interoperation feature translates addresses (through relation interTrans) between different domains. This is necessary because whenever a client from the feature's domain wishes to connect to a client attached to a different domain, it must have a target address it can use in its own domain space. Of course, the target client must have an address in each domain from which it is to be reached. Different facts are introduced in [11] in order to fully understand an interoperation feature behavior, and the following assertions are singled out:

- `ConnectedIsEquivalence`, asserting that field `connected` is indeed an equivalence relation (reflexive, symmetric and transitive).
- `UnidirectionalChains`, asserting that two hops are connected through a link in an ordered manner (one can be identified as *initiator* and the other one as *acceptor*).
- `Reachability`, asserting that whenever a client c publishes an address a in a domain d ($\langle a, d\rangle \in c.\mathtt{knownAt}$), clients c' from domain d can effectively connect to c.
- `Returnability`, asserting that if a client c accepted a connection from another client c', then a hop from c can be extended to a complete connection to client c'.

The Alloy description of the previous assertions is given in the Appendix due to the lack of space. We proved these properties from the Alloy model using Dynamite. Without using the modified pretty printer from PVS, the PVS specification of the returnability predicate looks like this:

```
FAL_Returnability :

  |-------
{1}   FORALL (hDm: (hop_domain), fDm: (feature_domain),
         tDm: (toDomain), tar: (target), rem: (remote),
         aCn: (atomConnected), con: (connected), oHp: (oneHop),
         aHp: (anotherHop), rBy: (reachedBy), map: (map),
         acc: (acceptor), srv: (servers), exp: (exported),
         imp: (imported), loc: (local), iTr: (interTrans),
         spc: (space), agn: (agent), oEd: (oneEnd),
         aEd: (anotherEnd), ini: (initiator),
         att: (attachments), src: (source)):
      FORALL (g1, g2: (Client), h1, h2, h3: (Hop)):
         Navigation_2(h1, ini)=g1 AND Navigation_2(h2, acc)=g2
         AND Leq(composition(composition(h1, one), h2),
                 Navigation(cConnections, con))
         AND Navigation_2(h3, ini)=g2
         AND Navigation_2(h3, hDm)=Navigation_2(h2, hDm)
         AND Navigation_2(h3, tar)=Navigation_2(h2, src)
         IMPLIES
         (EXISTS (h4: (Hop)):
            Navigation_2(h4, acc)=g1 AND
            Leq(composition(composition(h3, one), h4),
                 Navigation(cConnections, con)))
```

The modified pretty printer displays the same predicate to the user as follows:

```
FAL_Returnability :

  |-------
{1}   all g1,g2: Client, h1,h2,h3: Hop |
    (h1.ini)=g1 AND (h2.acc)=g2 AND
    (h1->h2) in (cConnections.con) AND
    (h3.ini)=g2 AND (h3.hDm)=(h2.hDm) AND (h3.tar)=(h2.src)
    IMPLIES
    (some h4: Hop |
        (h4.acc)=g1 AND (h3->h4) in (cConnections.con))
```

Notice that the pretty printed version closely resembles the Alloy definition. Furthermore, it can even be compiled with the Alloy Analyzer.

The proof fragment presented in the Appendix (Fig. 2) shows a branch in the proof tree of the property informally described as follows:

> *The "Accept" ends in a link, point to hops that contain the link's agent as acceptor.*

We have shown that it is possible to make proofs within the presented calculus with the aid of Dynamite. We now present some empirical data that will allow readers to have a better understanding of the usability of the tool.

The proofs were carried on by a student who had just graduated, and had no previous experience neither with Alloy, nor with PVS. The estimated time he spent in order to master the proof process is the following. 5 days to learn Alloy's syntax and semantics. 15 days to learn PVS, including the understanding of the proof rules. 40 days to prove all the assertions contained in the Alloy model. 15 days to prove the non trivial required lemmas about PDOCFAs. These lemmas can be considered as *infrastructure* lemmas, that will be reused in future proofs.

Recall that relations of rank greater than 2 are encoded as binary ones. Therefore, it may be necessary to prove properties that deal with the representation. These are the only proofs that would not be completely natural to an Alloy user. The proof of all the assertions in the model comprises 285 lemmas, of which only 12 use this kind of properties. Moreover, the 12 lemmas use actually 8 different properties of the representation because 3 properties are used at least twice.

Table 1 shows some numerical information about the proofs of the specific assertions. Notice that the sum of the total of lemmas amounts to 365. Therefore, $365 - 285 = 80$ lemmas were re-used in the proof of different assertions.

Assertion	Total Lemmas	Model Lemmas	Algebra Lemmas	Time (days)
ConnectedIsEquivalence	79	4	75	10
UnidirectionalChains	52	28	24	5
Reachability	121	62	59	23
Returnability	113	66	47	17

Table 1. Distribution of the workload.

6 Discussion

Abstracting from Alloy and PVS, our work can be described as a lightweight combination of a counterexample extractor with a semi-automatic theorem prover. This topic has been addressed by several researchers. Among the most relevant contributions we cite [7]. In [7], rather than focusing on providing theorem-proving capabilities to a lightweight formal method, the authors use model checking in order to look for counterexamples before (and during) the theorem proving process. This covers part (but not all) of our intentions when combining Alloy and PVS. In [10], alternative and more ambitious ways of combining model checking and theorem proving are presented. Model checkers and theorem provers interact using the latter for local deductions and propagation of known properties, while the former are employed in order to calculate new properties from reachability predicates or their approximations. Being Alloy models static, it is not clear how to employ these techniques, but it is clearly a road that we will explore in the near future. There are two approaches that we are aware of in what respects to theorem proving of Alloy assertions. One is the theorem prover Prioni [2]. Prioni translates Alloy specifications to first-order formulas characterizing their first-order semantics, and then the first-order logic theorem prover Athena [1] is used in order to prove the resulting theorem. While the procedure is sound, it is not completely amenable to Alloy users. Switching from a relational to a non relational language poses an overhead on the user. The other theorem prover is the one presented in [5]. This theorem prover translates Alloy specifications to a close relational language based on binary relations (the calculus for omega closure fork algebras [3]). Since the resulting framework is an equational calculus, quantifiers were removed from Alloy formulas in the translation process. This lead to very complicated equations, far from what an Alloy user would expect.

In this article we made two contributions. In the theoretical side we have provided a complete proof calculus for Alloy that we, as frequent Alloy users, find more amenable than previous ones. On the applied side we presented Dynamite, a tool that supports the interaction of the PVS semi-automatic theorem prover with the Alloy Analyzer. In order to assess the usability of the tool, we have proved several complex properties and obtained some empirical data. It is to expect that a domain expert used to the tool will make a more efficient use of Dynamite. This is a thesis we are testing by proving new network related properties recently supplied by Zave.

References

1. Arkoudas K., *Type-ω DPLs*, MIT AI Memo 2001-27, 2001.
2. Arkoudas K., Khurshid S., Marinov D. and Rinard M., *Integrating Model Checking and Theorem Proving for Relational Reasoning*, in Proceedings of RelMiCS'03 (Relational Methods in Computer Science), LNCS, Springer, 2003.
3. Frias M., *Fork Algebras in Algebra, Logic and Computer Science*, World Scientific Publishing Co., Series Advances on Logic, 2002.
4. Frias, M. F., Haeberer, A. M. and Veloso, P. A. S., *A Finite Axiomatization for Fork Algebras*, Logic Journal of the IGPL, Vol. 5, No. 3, 311–319, 1997.
5. Frias M.F., López Pombo C.G. and Aguirre N., *A Complete Equational Calculus for Alloy*, in Proceedings of Internacional Conference on Formal Engineering Methods (ICFEM'04), Seattle, USA, November 2004, Lecture Notes in Computer Science 3308, Springer-Verlag, 2004, pp. 162–175.
6. Jackson, D., Shlyakhter, I., and Sridharan, M., *A Micromodularity Mechanism.* Proc. ACM SIGSOFT Conf. Foundations of Software Engineering/European Software Engineering Conference (FSE/ESEC '01), Vienna, September 2001.
7. Kong W., Ogata K., , Seino T., and Futatsugi K., *A Lightweight Integration of Theorem Proving and Model Checking for System Verification*, in Proc. of APSEC'05, IEEE.
8. Maddux, R. D., *Pair-Dense Relation Algebras*, Transactions of the AMS, Vol. 328, N. 1, 1991.
9. Shankar N., Owre S., Rushby J. M., and Stringer-Calvert D. W. J., *PVS Prover Guide.* Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.
10. Shankar N., *Combining Theorem Proving and Model Checking through Symbolic Analysis*, in Proc. of CONCUR 2000, LNCS, Springer, 2000.
11. Zave, P., *A Formal Model of Addressing for Interoperating Networks*, in Proceedings of the Thirteenth International Symposium of Formal Methods Europe, Springer-Verlag LNCS 3582, pages 318-333, 2005.

A Appendix

```
assert ConnectedIsEquivalence { all c: Connections |
   (all h: Hop | h in h.(c.connected) ) &&
   (all h1,h2: Hop | h1 in h2.(c.connected) =>
                h2 in h1.(c.connected)) &&
   (all h1,h2,h3: Hop |
      h2 in h1.(c.connected) &&
      h3 in h2.(c.connected) =>
      h3 in h1.(c.connected) )
}

assert UnidirectionalChains { all l: Link |
      (l.agent = l.oneHop.acceptor &&
       l.agent = l.anotherHop.initiator) ||
      (l.agent = l.oneHop.initiator &&
       l.agent = l.anotherHop.acceptor)
}

assert Reachability{ all c: Connections,
   g1, g2: Client, h: Hop, a: Address, d: Domain |
      g1=h.initiator && d=h.domain && a=h.target &&
      (a->d) in g2.knownAt
   => (some h2: Hop | g2=h2.acceptor &&
                         (h->h2) in c.connected) }

assert Returnability{ all c: Connections,
   g1, g2: Client, h1, h2, h3: Hop |
   h1.initiator=g1 && h2.acceptor=g2 &&
   (h1->h2) in c.connected &&
   h3.initiator=g2 &&
   h3.domain=h2.domain && h3.target=h2.source
=> (some h4: Hop | h4.acceptor=g1 &&
                     (h3->h4) in c.connected) }
```

```
FAL_EveryAcceptorEndContainsTheAgentOfTheLink :

  |-------
{1}   all h: Hop, l: Link | (h=(l.oneHop) AND Accept=(l.oneEnd))
      OR (h=(l.anotherHop) AND Accept=(l.anotherEnd))
      IMPLIES (l.agent)=(h.acceptor)

Rule? (skosimp*)
Repeatedly Skolemizing and flattening, simplifies to:
FAL_EveryAcceptorEndContainsTheAgentOfTheLink :

{-1}  (h!1=(l!1.oneHop!1) AND Accept=(l!1.oneEnd!1)) OR
      (h!1=(l!1.anotherHop!1) AND Accept=(l!1.anotherEnd!1))
  |-------
{1}   (l!1.agent!1)=(h!1.acceptor!1)

Rule? (case "((l!1.oneEnd!1) in Accept
      IMPLIES (l!1.agent!1)=((l!1.oneHop!1).acceptor!1))
      AND ((l!1.anotherEnd!1) in Accept
      IMPLIES (l!1.agent!1)=((l!1.anotherHop!1).acceptor!1))")
Case splitting on
      ((l!1.oneEnd!1) in Accept IMPLIES
      (l!1.agent!1)=((l!1.oneHop!1).acceptor!1))
      AND ((l!1.anotherEnd!1) in Accept
      IMPLIES (l!1.agent!1)=((l!1.anotherHop!1).acceptor!1)),
this yields  2 subgoals:
FAL_EveryAcceptorEndContainsTheAgentOfTheLink.1 :

{-1}  ((l!1.oneEnd!1) in Accept IMPLIES
      (l!1.agent!1)=((l!1.oneHop!1).acceptor!1))
      AND ((l!1.anotherEnd!1) in Accept
      IMPLIES (l!1.agent!1)=((l!1.anotherHop!1).acceptor!1))
[-2]  (h!1=(l!1.oneHop!1) AND Accept=(l!1.oneEnd!1)) OR
      (h!1=(l!1.anotherHop!1) AND Accept=(l!1.anotherEnd!1))
  |-------
[1]   (l!1.agent!1)=(h!1.acceptor!1)

Rule? (flatten -1)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
FAL_EveryAcceptorEndContainsTheAgentOfTheLink.1 :

{-1}  ((l!1.oneEnd!1) in Accept IMPLIES
      (l!1.agent!1)=((l!1.oneHop!1).acceptor!1))
{-2}  ((l!1.anotherEnd!1) in Accept IMPLIES
      (l!1.agent!1)=((l!1.anotherHop!1).acceptor!1))
[-3]  (h!1=(l!1.oneHop!1) AND Accept=(l!1.oneEnd!1)) OR
      (h!1= (l!1.anotherHop!1) AND Accept=(l!1.anotherEnd!1))
  |-------
[1]   (l!1.agent!1)=(h!1.acceptor!1)

Rule? (split -3)
Splitting conjunctions, this yields 2 subgoals:
FAL_EveryAcceptorEndContainsTheAgentOfTheLink.1.1 :

{-1}  (h!1=(l!1.oneHop!1) AND Accept=(l!1.oneEnd!1))
[-2]  ((l!1.oneEnd!1) in Accept IMPLIES
      (l!1.agent!1)=((l!1.oneHop!1).acceptor!1))
[-3]  ((l!1.anotherEnd!1) in Accept IMPLIES
      (l!1.agent!1)=((l!1.anotherHop!1).acceptor!1))
  |-------
[1]   (l!1.agent!1)=(h!1.acceptor!1)

Rule? (hide -3)
Hiding formulas:  -3, this simplifies to:
FAL_EveryAcceptorEndContainsTheAgentOfTheLink.1.1 :

[-1]  (h!1=(l!1.oneHop!1) AND Accept=(l!1.oneEnd!1))
[-2]  ((l!1.oneEnd!1) in Accept IMPLIES
      (l!1.agent!1)=((l!1.oneHop!1).acceptor!1))
  |-------
[1]   (l!1.agent!1)=(h!1.acceptor!1)

Rule? (prop)
Applying propositional simplification, this yields 2 subgoals:
FAL_EveryAcceptorEndContainsTheAgentOfTheLink.1.1.1 :

{-1}  (l!1.agent!1)=((l!1.oneHop!1).acceptor!1)
{-2}  h!1=(l!1.oneHop!1)
{-3}  Accept=(l!1.oneEnd!1)
  |-------
[1]   (l!1.agent!1)=(h!1.acceptor!1)

Rule? (replace -2 -1 rl :hide? t)
Replacing using formula -2, this simplifies to:
FAL_EveryAcceptorEndContainsTheAgentOfTheLink.1.1.1 :

{-1}  (l!1.agent!1)=(h!1.acceptor!1)
[-2]  Accept=(l!1.oneEnd!1)
  |-------
[1]   (l!1.agent!1)=(h!1.acceptor!1)

which is trivially true.

This completes the proof of FAL_EveryAcceptorEndContainsTheAgentOfTheLink.1.1.1.
```

Fig. 2. A proof fragment.