Dynamite: A Tool for the Verification of Alloy Models Based on PVS[1]

Mariano M. Moscato,
Department of Computer Science, FCEyN, Universidad de Buenos Aires, Argentina.
and
Carlos G. Lopez Pombo,
Department of Computer Science, FCEyN, Universidad de Buenos Aires and CONICET, Argentina.
and
Marcelo F. Frias
Department of Software Engineering, Instituto Tecnológico de Buenos Aires (ITBA) and CONICET Argentina.

Automatic analysis of Alloy models is supported by the Alloy Analyzer, a tool that translates an Alloy model to a propositional formula that is then analyzed using off-the-shelf SAT-solvers. The translation requires user-provided bounds on the sizes of data domains. The analysis is limited by the bounds, and is therefore partial. Thus, the Alloy Analyzer may not be appropriate for the analysis of critical applications where more conclusive results are necessary.

Dynamite is an extension of PVS that embeds a complete calculus for Alloy. It also includes extensions to PVS that allow one to improve the proof effort by, for instance, automatically analyzing new hypotheses with the aid of the Alloy Analyzer. Since PVS sequents may get cluttered with unnecessary formulas, we use the Alloy unsat-core extraction feature in order to refine proof sequents. An internalization of Alloy's syntax as an Alloy specification allows us to use the Alloy Analyzer for producing witnesses for proving existentially quantified formulas.

Dynamite complements the partial automatic analysis offered by the Alloy Analyzer with semi-automatic verification through theorem proving. It also improves the theorem proving experience by using the Alloy Analyzer for early error detection, sequent refinement and witness generation.

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

General Terms: Design, Verification

Additional Key Words and Phrases: Alloy, PVS, Alloy Calculus, Unsat-cores

[1]This publication was made possible by NPRP grant NPRP-4-1109-1-174 from the Qatar National Research Fund (a member of Qatar Foundation). The statements made herein are solely the responsibility of the authors.

E–mails: mmoscato@dc.uba.ar, clpombo@dc.uba.ar, mfrias@itba.edu.ar.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0000-0000/20YY/0000-0001 $5.00

1. INTRODUCTION

There is a vast spectrum of tools and techniques for software analysis. If we just concentrate on the degree of automation offered by these tools and techniques, the spectrum goes from fully automatic (lightweight) to fully user-driven (heavyweight). Full automation has its price, often paid by limitations on the kind of analysis provided, or by the lack of scalability of the analysis. Model checking [Clarke et al. 1999] using tools such as SPIN [Holzmann 2003] and SAT-solving using, for instance, the Alloy Analyzer [Jackson 2006] are examples of methods that lay on this side of the spectrum. User guidance offers more conclusive analysis techniques, but requires trained users. Also, user-guided techniques are often time consuming. Tools based on theorem provers (PVS [Owre et al. 1992], Isabelle [Nipkow et al. 2002] or Coq [Bertot et al. 2004]), require a significant effort from the user.

If we confine ourselves to the analysis technique of determining if a model (specification) is appropriate by analyzing whether certain given properties hold in the model, then both model checking and SAT-solving can be used. In both cases, the techniques have limitations on the expressiveness of the languages they can analyze. This is particularly clear in the case of SAT-solving, where the source language is propositional logic. Also, most model checkers support decidable fragments of logics. If we instead consider theorem proving, more expressive languages can be used. For instance, there are many complete theorem provers for classical first-order logic. These include techniques for automatically proving “easy” properties, but more complex ones require creative steps that must be guided by a human user. Proving a theorem can be a difficult and tedious activity. Even more so if incorrect hypotheses are introduced along the way. This situation, which is more common than it is desirable, is both discouraging and time consuming. Notice that every proof step that depends on the wrong hypotheses has to be reconsidered when developing a new proof.

Despite their limitations, automatic and user-driven methods also have qualities that make them valuable. If we manage to translate our model to a propositional formula (in the case of SAT-solving) or to an automaton (in the case of model checking), the (probably weaker) model thus obtained can be automatically analyzed. Notice that the result of the analysis of the weaker model can offer partial information about the original model. If a complete calculus for our specification language exists, then there is no need to translate our model to a weaker one. Proving a property using such calculus allows us to conclude its validity in the model.

As an example, let us consider the Alloy modeling language [Jackson et al. 2001]. Alloy is a formal specification language that allows one to create data domains, and express properties of relations defined over those domains. We will present Alloy in Section 2, but we point out here that since Alloy extends classical first-order logic (it even includes reflexive-transitive closure of binary relations, which is not expressible in classical first-order logic), it is not decidable. The Alloy Analyzer [Jackson 2006] is a tool that allows one to automatically analyze Alloy models by searching for counterexamples for a given property using off-the-shelf SAT-solvers. The impedance mismatch between the undecidable Alloy language and the decidable language on which SAT-solvers operate, is clear. The gap is bridged by translating Alloy models to propositional models. The translation does not come for free. It

requires users to provide bounds (called *scopes* in the Alloy terminology) on the size of data domains. The bounded model is the one translated. The result of the SAT-based analysis is then valid for those semantic structures whose data domains are constrained by the chosen scopes, i.e., if a counterexample is not found, one might still exist if larger scopes were chosen. While this analysis technique has obvious limitations, it is nevertheless very useful when creating a model. Although possible, it is seldom the case that errors introduced in a model can only be exhibited within large models. The *small scope hypothesis* [Andoni et al. 2002] even claims that most software errors can be made explicit by resorting to small domain sizes. Therefore, many errors introduced when building a model can be discovered by performing bounded analysis using small bounds.

For many modeling situations the kind of analysis offered by the Alloy Analyzer is entirely satisfactory. In some cases, however, this analysis may fall short. This is evident when building models for critical systems. Knowing that no small errors exist is not enough. Therefore, in this article we propose to extend the SAT-solving analysis provided by the Alloy Analyzer with a user-guided theorem prover based on PVS.

In order to prove in PVS that a set of formulas $\Delta = \{ \delta_1, \ldots, \delta_m \}$ follows from a set of hypotheses $\Gamma = \{ \gamma_1, \ldots, \gamma_k \}$, one begins with the *sequent* $\Gamma \vdash \Delta$. Applying inference rules, from $\Gamma \vdash \Delta$ one must reach other sequents that can be recognized as valid (for example, sequents of the form $\alpha \vdash \alpha$). The informal understanding of the sequent $\Gamma \vdash \Delta$ is that, from the conjunction of the formulas in Γ, the disjunction of the formulas in Δ must follow. The formulas in Γ (Δ) are called the *antecedent* (*consequent*) of the sequent.

When an inference rule is applied on a sequent S, new sequents $S_1, \ldots, Sn$ are produced. Proving sequent S then reduces to finding proofs for the intermediate sequents $S_1, \ldots, Sn$. Our experience as users of PVS is that, when proving a given sequent, the number of antecedents and consequents in intermediate sequents tends to grow. This leads many times to sequents containing formulas that are unnecessary for their proof. These formulas make the identification of new proof steps more complex. PVS provides a command (`hide`) for hiding hypotheses and conclusions in sequents, yet its incorrect use may lead to hiding necessary antecedents or consequents, making the proof infeasible.

Alloy extends first-order logic. While dealing with propositional connectives is straightforward, quantifiers pose a challenge. Particularly complex is the problem of finding witnesses when attempting to prove existentially quantified assertions.

1.1 Contributions

The contributions of this article can then be summarized as follows:

—We present a complete proof calculus for the Alloy modeling language.

—We extend the PVS theorem prover so that Alloy assertions can be proved using Alloy syntax.

—We facilitate the interaction between PVS and the Alloy Analyzer in order to reduce the number of theorem proving errors induced by introduction of erroneous hypotheses, introduction of false lemmas, or hiding of necessary hypotheses.

—We present a heuristic to reduce the proof search space based on the use of

UnSAT cores to remove possibly unnecessary antecedents and consequents in a sequent. The technique also allows us to remove formulas from the underlying theories being considered.

—We present a novel technique based on SAT for automatic generation of witness candidates for existentially quantified Alloy assertions.

—We present several examples, including a complex case-study coming from the networking domain [Zave 2006], that allow us to assess the usefulness and usability of Dynamite

1.2 Related Work

Some of the goals we envisioned when we started this article were previously addressed either by colleagues or ourselves. There are two approaches previous to ours in what respects to theorem proving of Alloy assertions. One is the theorem prover Prioni [Arkoudas et al. 2004]. Prioni translates Alloy specifications to first-order formulas characterizing their first-order semantics, and then the first-order logic theorem prover Athena [Arkoudas 2001] 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 that we are trying to reduce as much as possible. The other theorem prover is the one presented in [Frias et al. 2004]. This theorem prover translates Alloy specifications to a close relational language based on binary relations (the calculus for omega closure fork algebras [Frias 2002]). Since the resulting framework is an equational calculus, quantifiers are removed from Alloy formulas in the translation process. This led to complicated equations, unnatural for standard Alloy users. In our articles [Frias et al. 2007; Moscato et al. 2010], we announced some of the contributions made in this article. There are differences though, namely:

—We present a proof of completeness of the calculus we propose, including definitions that are more precise than those in [Frias et al. 2007].

—We present a different case study, explained in greater detail.

—We discuss new features incorporated to Dynamite. In particular, we consider a new command for *introduction of cases* (c.f. Section 5.3), and a new command for *hiding of sequent formulas* that analyzes whether the hidden formulas from a sequent were indeed superfluous (c.f., Section 5.4).

—We present a novel technique that allows us to automatically generate witness candidates for existentially quantified assertions.

More recent articles [El Ghazi et al. 2011; Ulbrich et al. 2012] also address the problem of verifying Alloy assertions. Article [El Ghazi et al. 2011] translates the Alloy model to an SMT-problem, which is solved using the SMT-solver Z3 [de Moura et al. 2008]. This is a very interesting approach that has some limitations. Complex declarative assertions (as the ones we deal with in this article) are unlikely to be solved automatically. Also, the experimental results show that spurious counterexamples can be produced. Article [Ulbrich et al. 2012] follows an approach close to ours. It presents the Kelloy prover, which is built on top of the KeY first-order theorem prover [Beckert et al. 2007]. Kelloy's embedding into

KeY seems to provide greater automation, but no integration with the Alloy Analyzer. In particular, a limitation mentioned in the article is the need for quantifier instantiation. For some of the examples used in [Ulbrich et al. 2012], the witness generation technique we introduce here was able to produce correct instantiations, automatically. Another difference arises in the way integers are modeled. While [El Ghazi et al. 2011; Ulbrich et al. 2012] depart from the Alloy semantics by considering the standard mathematical model of the integers, we stick to the Alloy model where a 2's-complement representation of integers is considered (c.f. Section 3.4). Nitpick [Blanchette et al. 2010] is used as a counterexample generator for the higher-order logic supported by the Isabelle theorem prover. Nitpick, like Dynamite, aims at detecting non-theorems when a proof is initiated. Unlike Dynamite, Nitpick's application seems to be restricted to this case.

1.3 Organization

The article is organized as follows. In Section 2 we present a brief introduction to the Alloy modelling language and the Alloy Analyzer. In Section 3 we present the complete calculus for Alloy that will be made accessible through our tool, Dynamite. In Section 4 we describe the way in which the complete calculus presented in Section 3 is embedded in PVS, as well as discuss important implementation details. In Section 5 we describe the features of Dynamite through several case studies. Finally, in Section 6 we present some conclusions and discuss further improvements to the tool.

2. AN INTRODUCTION TO ALLOY

In this section we describe the Alloy modeling language and the Alloy Analyzer with the level of detail required in order to follow this article. For a thorough description we point the reader to the book [Jackson 2006], which provides numerous examples of varied complexity illustrating Alloy's features. In Fig. 1 we present a sample Alloy model. The model describes a simple file system, where files are stored in directories accessible from a root directory. Models are organized in *modules*. Signatures (identified with the keyword *sig*) denote data domains. A signature can be *abstract* (as is the case of signature `Object`), in which case it will only hold elements from inheriting signatures. (Single) inheritance is distinguished in Alloy with the keyword *extends*. For example, signature `Dir` extends `Object`. Signatures may contain attributes, which are called *fields* in Alloy. Field `contents` denotes a ternary relation contained in `Dir` $\times$ `Name` $\times$ `Object`. The modifier `lone` in the declaration of field `contents` actually relates `Dir` objects with partial functions from `Name` to `Object`. Similarly, field `parent` denotes a binary relation contained in `Dir` $\times$ `Dir`. The modifier `lone` constrains this relation to be a partial function. A *one sig* can have exactly one element in the denoted set (see for instance signature `Root`). Once domains are defined, constraints can be added. Axioms are called *facts* in Alloy terminology. In order to simplify writing facts, functions (noted `fun`) and predicates (noted `pred`) can be introduced. Terms denote relations, and are built from signatures, signature fields and constants such as `univ` (set of all objects in the model), `iden` (identity binary relation on the set `univ`), and `none` (the empty set). Relational operators are used to build more complex terms. Difference, union, intersection and composition of relations are denoted by $-$, $+$, & and $\cdot$, re-

```
module file_system

abstract sig Object {}

sig Name {}

sig File extends Object {}

sig Dir extends Object {
  contents: Name -> lone Object,
  parent: lone Dir }

one sig Root extends Dir {}

pred parentIsWellDefined[d: Dir] { d.parent = (contents.d).Name }

fact ParentDefinition { all d: Dir | parentIsWellDefined[d] }
fact RootHasNoParent { all r: Root | no r.parent  }

fun ancestors[d: Dir]: Dir { d.^parent }

fact NoOwnAncestor { all d: Dir | d !in ancestors[d] }
fact RootIsTheRoot { all d: Dir - Root | Root in ancestors[d] }
fact OneParent { all d: Dir | d !in Root =>
      (one d.parent && one contents.d) }

assert NoDirAliases { all o: Dir | lone (contents.o) }
check NoDirAliases for 4
```

Fig. 1. An Alloy sample model.

spectively. The transpose of a binary relation (denoted by $\sim$) flips the elements in pairs, around. Transitive closure and reflexive-transitive closure of binary relations are denoted by $\hat{}$ and $*$, respectively. Formulas are built from equalities and inclusions between terms using the standard connectives and quantifiers from first-order predicate logic. Figure 2 presents the formal syntax and semantics of Alloy terms and formulas. Abbreviations are used to simplify formulas. For example, keyword `no` in fact `RootHasNoParent` constrains term `r.parent` to denote an empty relation. Similarly, keyword `one` in fact `OneParent` implies that the term denotes a singleton relation. A keyword `lone` constrains a term to denote a set whose cardinality is at most 1.

Assertions are sentences that are expected to hold in the model as a means to validate the model correctness. The Alloy Analyzer appropriately translates the model and the (negation of the) assertion to a propositional formula, and an off-the-shelf SAT-solver is used in order to look for a satisfying valuation. If such valuation is found, a counterexample for the assertion can be retrieved from it. *Check* commands tell the Analyzer about the sizes of domains (scopes) to be used during analysis. For instance, the check command in Fig. 1 states that up to 4 objects in each domain can be used during analysis.

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

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

expr ::=
iden (identity relation)
| *univ* (universe set)
| *none* (empty set)
| expr + expr (union)
| expr & expr (intersection)
| expr − expr (difference)
|∼ expr (transpose)
| expr.expr (navigation)
| ∗expr (closure)
| *Var*

X : expr $\rightarrow$ *env* $\rightarrow$ *value*
$X[iden] = \{\,\langle a, a\rangle : a \text{ is an atom}\,\}$
$X[univ] = \{\,a : a \text{ is an atom}\,\}$
$X[none] = \emptyset$
$X[a + b]e = X[a]e \cup X[b]e$
$X[a\&b]e = X[a]e \cap X[b]e$
$X[a - b]e = X[a]e \setminus X[b]e$
$X[\sim a]e = \{\,\langle x, y\rangle : \langle y, x\rangle \in X[a]e\,\}$
$X[a.b]e = X[a]e.X[b]e$
$X[*a]e = \text{smallest } r \text{ s. t. } iden \subseteq r,$
$\quad r.r \subseteq r \text{ and } X[a]e \subseteq r$
$X[v]e = e(v)$

Fig. 2. Alloy's syntax and semantics.

3. A COMPLETE CALCULUS FOR ALLOY

In this section we present a deductive calculus useful for the verification of Alloy assertions. The procedure we will use for presenting the calculus is the following:

—We will introduce the class of "proper point-dense omega closure fork algebras". These algebras contain operations akin to Alloy operations. We will also present a complete calculus for this class of algebras. The deduction relation in this formalism will be denoted by $\vdash$.

—We will present an interpretability theorem from Alloy theories to fork algebra theories. An interpretability theorem consists of a mapping $F : \mathit{AlloyForm} \rightarrow \mathit{ForkForm}$ (mapping Alloy formulas to fork formulas), and a theorem proving that:

$$\Gamma \models_{\text{Alloy}} \alpha \quad \iff \quad \{\,F(\gamma) : \gamma \in \Gamma\,\} \vdash F(\alpha)\ .$$

—We show how Alloy integers and cardinality are modeled in the proposed formalism, and discuss the consequences of adopting such model.

Notice that checking the validity of an Alloy assertion in a specification then reduces to the problem of proving a property in the deductive calculus of fork algebras. Since the fork algebraic formalism is not exactly Alloy, it is essential to discuss to what extent is the new formalism useful for Alloy users. This discussion permeates Sections 3.1–3.3.

In Section 3.1 we present the fork formalism. In Section 3.2 we discuss how Alloy quantification is modeled in a formalism where quantifiers range over relations. In

Section 3.3, we present the interpretability result. Notice that a particular theory that has to be interpreted in the algebraic formalism is the Alloy theory for integers (c.f. Section 3.4).

3.1 Point-Dense Omega Closure Fork Algebras

We begin this section by introducing the class of *proper* point-dense omega closure fork algebras. Qualifier “proper” refers to the fact these algebras are special in the sense that they are particularly close to the semantics of Alloy. In effect, these algebras have (binary) relations (on a given set B) in their universe, and operations for union, intersection, difference[2], navigation, transposition and closure of relations, as Alloy has.

Definition 3.1. A *proper* point-dense omega closure fork algebra on a set B is a structure $\langle\, R, +, \&, ^{-}, \emptyset, \mathit{univ}, ., \mathit{iden}, \sim, *, \nabla \,\rangle$,

where:

— R is a set of binary relations on the set B, closed under the operations.
— $^{-}$ is set-complement, $+$ is set-union and $\&$ is set-intersection.
— $\emptyset$ is the empty set, and *univ* is the binary relation $B \times B$.
— . is composition (called *navigation* in Alloy terminology) between binary relations.
— *iden* is the identity relation on B.
— $\sim$ is transposition of binary relations.
— $*$ is reflexive-transitive closure of binary relations (and $\hat{}$ is transitive closure).
— ∇ is the *fork* operation. It is defined as

$$S \nabla T = \{\, \langle a, b \star c \rangle : \langle a, b \rangle \in S \ \wedge \ \langle a, c \rangle \in T \,\} \ . \tag{1}$$

Symbol $\star$ in (1) stands for an injective function of type $B \times B \to B$. Therefore, we assume set B to be closed under $\star$.

Notice that while function $\star$ has to be injective, it may not be surjective. Therefore, there may exist elements in B that do not encode pairs. These elements are called *urelements*.

We also constrain these algebras to be “point-dense” [Maddux 1991]. 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 $p \subseteq I$.

There are essentially two ways in which the previous definition departs from Alloy, namely, the existence of the fork operator (not directly tied to any Alloy operator), and the request for point-density. As we will see in Section 3.3, these are essential in order to provide the complete calculus; fork will be necessary in order to emulate relations of arity greater than two in a calculus that only handles binary relations, and point-density is required in order to define Alloy’s quantifiers.

[2]Actually, these algebras have a complement operation, but the latter allows us to define difference with the aid of intersection.

We now introduce a larger class of algebras as the class of models of a finitely axiomatized theory. These algebras, called point-dense omega closure fork algebras (we will denote the class by PDOCFA) are closely related (as will be shown in Thm. 3.3) to their "proper" counterpart. In order to present the theory we will present the axioms and the proof rules. Before doing so, we introduce some notation.

NOTATION 1. In a proper PDOCFA the relations π and ρ defined by

$$\pi = \sim (iden \nabla univ) \quad \text{and} \quad \rho = \sim (univ \nabla iden)$$

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

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

The binary operation *cross* (denoted by $\otimes$) 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 definable with the aid of fork via the equation

$$r \otimes s = (\pi . r) \ \nabla \ (\rho . s) \ .$$

We can characterize points as nonempty binary relations that satisfy the property $x . univ . x \subseteq iden$. If we denote the inclusion relation by "in" (as in Alloy), the predicate "Point" defined by

$$\text{Point}(p) \iff p \mathrel{!=} \emptyset \ \&\& \ p . univ . p \text{ in } iden$$

characterizes those relations that are points.

The axioms and inference rules for the calculus are given in the following definition.

DEFINITION 3.2. The calculus for point-dense omega closure fork algebras is characterized by the following axioms and proof rules:

(1) Axioms for Boolean algebras characterizing the meaning of $+$, $\&$, $^-$, $\emptyset$ and *univ*.
(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 $(\sim x . z) \& y = \emptyset$,
$*x = iden + (x . *x)$,
$*x . y . univ$ in $(y . univ) + \left(*x . (\overline{y . univ} \ \& (x . y . univ))\right)$.
(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 | $(x \mathrel{!=} \emptyset \ \&\& \ x$ in $iden) \Rightarrow ($some p | Point$(p) \ \&\& \ p$ in $x)$,

(5) Term $\overline{univ\,.\,(univ\,\nabla\,univ)}$ & $iden$ (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:
$univ\,.\,iden_{\mathsf{U}}\,.\,univ = univ.$

The inference rules for the closure fork calculus are those for classical first-order logic (choose your favorite ones), plus the following equational (but infinitary) proof rule for reflexive-transitive closure (given $i > 0$, by x^i we denote the relation inductively defined as follows: $x^1 = x$, and $x^{i+1} = x\,.\,x^i$):

$$\frac{\vdash iden \text{ in } y \quad x^i \text{ in } y \vdash x^{i+1} \text{ in } y}{\vdash *x \text{ in } y} \ (\omega\!-\!Rule)$$

The axioms and rules given above define a class of models. Proper PDOCFA belong to this class, but there might be models for the axioms that are not proper PDOCFA. Fortunately, the following theorem (which follows from [Frias et al. 1997], [Frias 2002, Thm. 4.2] and [Maddux 1991, Thm. 52]), states that if a model is not a proper PDOCFA, then it is isomorphic to one.

THEOREM 3.3. *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* $iden = \{\langle a_0, a_0\rangle, \ldots, \langle a_i, a_i\rangle, \ldots\}$.

Notice that the previously presented calculus and classes of algebras share most of the operations with Alloy (at least intentionally and notationally). While Thm. 3.3 is important in itself (it implies that the calculus is complete with respect to the properties valid in proper PDOCFAs), it is necessary in order to prove theorems on the appropriateness of the deductive mechanism we will provide for Alloy in Section 3.3. There is a gap between the language of PDOCFA and Alloy that still needs to be bridged, namely, the way quantification is defined. While quantification in Alloy ranges over atomic elements from signatures, in PDOCFA, quantifiers range over all the relations in the domain. In Section 3.2 we will introduce appropriate notation in the language of PDOCFA in order to bridge the gap. For a thorough discussion about fork algebras and their applications, the reader is directed to [Frias 2002].

3.2 Constraining PDOCFA Quantifiers to Points

When we write an Alloy formula such as

```
all d, d' : Domain | d.space != d'.space => d != d',
```

quantified variables d and d' range over `Domain` objects. There is a single signature in PDOCFA, namely, the one that holds all the relations. Therefore, given an algebra in PDOCFA, quantifiers range over all the relations in the domain of the algebra. Therefore, while the Alloy operations have an almost direct counterpart in PDOCFA, quantified formulas do not. This is when points and point-density (as presented in Def. 3.2) become necessary. Recall that a point is a relation that in proper PDOCFAs has the form $\{\langle a, a\rangle\}$. We then associate an Alloy singleton $\{a\}$ with the point $\{\langle a, a\rangle\}$. Moreover, we will associate Alloy signatures with partial identities in PDOCFA. While we will present the details of the translation in Section 3.3 (c.f. Def. 3.8), an Alloy signature A will be represented by a partial identity

$iden_A$ satisfying $iden_A = \{ \langle a, a \rangle : a \in A \}$. We can then map an Alloy formula of the form

```
all a : A | Form
```

to a PDOCFA formula of the form

```
all a | (Point(a) && a in iden_A) => Form'
```

where `Form'` is the translation of formula `Form`.

Similarly, an existentially quantified Alloy formula

```
some a : A | Form
```

will be mapped to the formula

```
some a | Point(a) && a in iden_A && Form'
```

In order to retain the similarity between Alloy formulas and their counterparts, we will introduce the following notation:

```
all a | (Point(a) && a in iden_A) => Form'
```

will be abbreviated to

```
all a : A | Form'
```

and similarly, formula

```
some a | Point(a) && a in iden_A && Form'
```

will be abbreviated to

```
some a : A | Form'
```

Notice that the above abbreviations equate (up-to translation of terms) the source Alloy formulas and their translation to PDOCFA.

3.3 Interpretability of Alloy in PDOCFA

One of the (main) goals of this article is to present a complete deductive mechanism for Alloy. In order to fulfill this task we will prove an interpretation theorem of Alloy theories as PDOCFA theories. An interpretation theorem of Alloy in PDOCFA (as described in the introduction to Section 3), allows us to map semantic entailment in Alloy to deductions in PDOCFA. Such result allows us to use the calculus for PDOCFA in the following way. If we want to prove a given assertion α in an Alloy model (specification) M, we translate α and M to PDOCFA formulas α' and M', and prove that α' follows from M' according to the PDOCFA calculus. As we discussed in Section 1, the idea of mapping Alloy to an expressive-enough formalism in which to carry proofs on is not entirely new. It has already been done for instance with Prioni [Arkoudas et al. 2004]. The (essential) advantage of the mapping we propose in this article is that the resulting formalism is extremely close to Alloy, and therefore easier to grasp by standard Alloy users. While this feature may be useless in the context of fully automated tools (for which the language target of the translation may be ignored by the user), it is of utter importance for user-guided

tools. In the remaining parts of this section we give a proof of the interpretability theorem.

The main part of an interpretability theorem is a mapping from Alloy formulas to formulas in the language of PDOCFA. The mapping is defined in two stages, since Alloy terms must be mapped as well. We will present maps T (mapping terms), and F (mapping formulas). Mapping F's definition uses T. Mapping T must, in particular, translate relational constants coming from Alloy signatures. Since these are interpreted in Alloy as n-ary relations, and PDOCFA only deals with binary relations, we must find an adequate means for modeling relations of arity greater than 2 as binary relations. The operator fork allows us to do this in a simple way. An invariant behind the translation is that given an Alloy unary relation, it is translated to a binary partial identity (i.e., a binary relation contained in the identity relation). For relations of arity greater than (or equal to) 2 holding tuples of the form $\langle a_1, a_2, \ldots, a_n \rangle$, the binary relation resulting from the translation will hold tuples of the form[3] $\langle a_1, a_2 \star \cdots \star a_n \rangle$. Although the resulting relation is binary, we will say its rank is n if it encodes an Alloy relation of rank n. Notice that navigation in PDOCFA must be modified in order to behave as expected for Alloy. We define in PDOCFA a new operation denoted by $\bullet$ that preserves the previously given invariant.

Definition 3.4. Given a binary relation R, by $Dom\,(R)$ we denote the partial identity over the elements in R's domain. Similarly, by $Ran\,(R)$ we denote the partial identity over the elements in R's range. Since we are encoding n-ary Alloy relations as binary ones, by rank(R) (R being the binary relation) we denote the arity of the original Alloy relation. In algebraic terms, we have

$$Dom\,(R) = (R\ .\ \sim R)\,\&iden, \quad \text{and} \quad Ran\,(R) = (\sim R\ .\ R)\,\&iden\ .$$

$$R \bullet S = \begin{cases} Ran\,(R.S) & \text{if rank}(R) = 1 \wedge \text{rank}(S) = 2 \\ \sim \pi\,.\,Ran\,(R.S)\,.\rho & \text{if rank}(R) = 1 \wedge \text{rank}(S) > 2 \\ Dom\,(R.S) & \text{if rank}(R) = 2 \wedge \text{rank}(S) = 1 \\ R.S & \text{if rank}(R) = 2 \wedge \text{rank}(S) > 1 \\ R.\,(iden \otimes (iden \otimes (\cdots \otimes ((iden \otimes S)\,.\pi)))) & \text{if rank}(R) > 2 \wedge \text{rank}(S) = 1 \\ R.\,(iden \otimes (iden \otimes (\cdots \otimes (iden \otimes S)))) & \text{if rank}(R) > 2 \wedge \text{rank}(S) > 1 \end{cases} \tag{2}$$

Let us illustrate Eq. (2) with an example. Relation `contents` (from signature `Dir` introduced in Fig. 1), is ternary. We will assume that for each relational constant in an Alloy model (be it a signature or a signature field), there is a corresponding binary constant added to the language of PDOCFA. We will use the following notational convention.

Notation 2. If an Alloy signature is called S, the PDOCFA constant we add is named $iden_S$. Similarly, given an Alloy field named X, the PDOCFA counterpart will be denoted by X.

Then, relation contents satisfies

$$\mathsf{contents} = \{\ \langle a, b \star c \rangle : \langle a, b, c \rangle \in \texttt{contents}\ \}\ .$$

[3]Since $\star$ is not associative, an expression of the form $a \star b \star c$ denotes the object $a \star (b \star c)$.

If `ob` is an object atom (i.e., a unary Alloy relation of the form $\{\, ob \,\}$ for some ob from signature `Object`), the navigation `contents.ob` produces as a result a binary relation contained in `Dir` $\times$ `Name`. Let us analyze what is the result of applying $\bullet$ on the PDOCFA representation of contents and ob. We obtain:

$$
\begin{aligned}
& \mathsf{contents} \bullet \mathsf{ob} \\
= & \text{ (by Def. of } \bullet) \\
& \mathsf{contents}.(iden \otimes \mathsf{ob}).\pi \\
= & \text{ (by Def. of ``.'')} \\
& \{\langle d, a\rangle : \text{some } x : \mathtt{Name}, y : \mathtt{Object} \mid \langle d, x \star y\rangle \text{ in } \mathsf{contents} \wedge \\
& \quad \text{some } x' : \mathtt{Name}, y' : \mathtt{Object} \mid \langle x \star y, x' \star y'\rangle \text{ in } iden \otimes \mathsf{ob} \wedge \\
& \quad \langle x' \star y', a\rangle \in \pi\} \\
= & \text{ (because } \langle x, x'\rangle \in iden \wedge \langle y, y'\rangle \in \mathsf{ob} \subseteq iden) \\
& \{\langle d, a\rangle : \text{some } x : \mathtt{Name}, y : \mathtt{Object} \mid \langle d, x \star y\rangle \text{ in } \mathsf{contents} \wedge \\
& \quad \langle y, y\rangle \in \mathsf{ob} \wedge \langle x \star y, a\rangle \in \pi\} \\
= & \text{ (by Def. of } \pi, \text{ is } x = a) \\
& \{\langle d, a\rangle : \text{some } y : \mathtt{Object} \mid \langle d, a \star y\rangle \text{ in } \mathsf{contents} \wedge \langle y, y\rangle \in \mathsf{ob}\} \\
= & \text{ (due to the relationship between } \mathsf{contents} \text{ and } \mathtt{contents}, \text{ and } \mathsf{ob} \text{ and } \mathtt{ob}) \\
& \{\langle d, a\rangle : \text{some } y : \mathtt{Object} \mid \langle d, a, y\rangle \text{ in } \mathtt{contents} \;\&\&\; y \in \mathtt{ob}\} \\
= & \text{ (because } \mathtt{ob} \text{ is an atom from } \mathtt{Object}) \\
& \{\langle d, a\rangle : \langle d, a, \mathtt{ob}\rangle \text{ in } \mathtt{contents}\}
\end{aligned}
$$

It is worth emphasizing that, while Alloy formulas and PDOCFA formulas are close, still there are differences between the formalisms. In Def. 3.5 we will present the translation for Alloy terms. As it was discussed in Section 3.1, operator ∇ will be necessary in order to emulate relations of arity greater than two in a calculus that only handles binary relations. We introduced in Eq. (2) an operator $\bullet$ (that uses ∇ in its definition) that interprets composition between Alloy relations whose arity may be different from 2. The differences from the Alloy language show when we need to prove properties of $\bullet$ that require using the underlying fork algebra definition that includes the operator ∇. We expect the number of properties involving the definition of $\bullet$ to be small compared to the complete proof. In the case study we are reporting, 25 out of 60 proved lemmas required dealing with the low-level representation of $\bullet$. Yet 18 out of these 25 properties relate to properties of $\bullet$ that may be reused along other proofs. For example, among these 18 properties the following general properties are included:

```
all A:set univ, B:set univ, R:set(A->B), S:set(A->B), a:A |
   R in S implies a.R in a.S
all A:set univ, B:set univ, R:set(A->B), S:set(A->B) |
   (R.B)+(S.B) = (R+S).B
all A:set univ, B:set univ, C:set univ, W:set(A->(B->C)), a:A |
   a.W in B->C
```

We include a library with those properties of $\bullet$ that we consider general and useful.

The following definition introduces the mapping for terms.

Definition 3.5. Function T maps Alloy terms to expressions in the language of PDOCFA (x_i is a variable ranging over Alloy objects, and X_i is a variable ranging

over points).

$$
\begin{array}{ll}
T(x_i) = X_i, & T(iden) = iden_{\mathsf{U}}, \\
T(univ) = iden_{\mathsf{U}}, & T(none) = \emptyset, \\
T(sig_i) = iden_{sig_i}, & T(C) = \mathsf{C}, (C \text{ is an Alloy field}) \\
T(\sim r) = \sim T(r), & T(*r) = *T(r), \\
T(r+s) = T(r) + T(s), & T(r\&s) = T(r) \ \& \ T(s), \\
T(r-s) = T(r) \ \& \ \overline{T(s)}, & T(r.s) = T(r) \bullet T(s).
\end{array}
$$

Formal semantics of Alloy is defined in terms of an *environment*. An environment is a function that assigns sets to signatures, adequately typed relations to relational constants (those arising from signature fields), and values to variables over individuals. The translation process requires adding new constant relations to PDOCFA corresponding to the relational constants in the Alloy model. We have already explained how signatures and fields are modeled in the algebraic setting. Regarding individual Alloy variables, our convention is that these are modeled using relational variables ranging over points. From an Alloy environment we can build a PDOCFA environment assigning meaning to these new constants. The construction is done as follows.

Definition 3.6. Given an Alloy environment e we define a PDOCFA environment e_{P} (P stands for PDOCFA), as follows:

—If S is a signature, we define $e_{\mathsf{P}}(iden_S) = \{ \langle s, s \rangle : s \in e(S) \}$,

—If F is an n-ary field ($n \geq 2$), then

$$e_{\mathsf{P}}(\mathsf{F}) = \{ \langle a_1, a_2 \star \cdots \star a_n \rangle : \langle a_1, a_2, \ldots, a_n \rangle \in e(F) \},$$

—If v is a variable (ranging over Alloy atoms), then (recall that the corresponding relational variable ranging over points is noted V) $e_{\mathsf{P}}(V) = \{ \langle e(v), e(v) \rangle \}$. Notice that the resulting relation is indeed a point.

Similarly, given a proper PDOCFA and a relational environment we can define a sort of canonical Alloy environment.

Definition 3.7. Let $\mathfrak{F}$ be a proper PDOCFA and let e be a relational environment assigning meaning to constans in $\mathfrak{F}$. We define an Alloy environment e_{A} (A stands for Alloy) as follows:

—$e_{\mathsf{A}}(\mathrm{sig}_i) = \left\{ a : \langle a, a \rangle \in e(iden_{\mathrm{sig}_i}) \right\}$,

—$e_{\mathsf{A}}(R) = \{ \langle a_1, \ldots, a_n \rangle : \langle a_1, a_2 \star \cdots \star a_n \rangle \in e(\mathsf{R}) \}$,

—$e_{\mathsf{A}}(v_i) = a$ such that $e(V_i) = \{ \langle a, a \rangle \}$.

Once the translation of terms has been presented, we introduce the translation from Alloy formulas to PDOCFA formulas. The translation differs from the one presented in [Frias et al. 2004] 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 they are kept explicit. This will greatly improve the readability of the translated formulas by Alloy users.

DEFINITION 3.8. Function F, defined below, maps Alloy formulas to PDOCFA formulas.

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

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

We then prove the following completeness theorem (recall that the turnstile symbol $\vdash$ notes the derivability relation in the calculus of PDOCFAs).

THEOREM 3.9. *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. 3.3 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 A.6, there exists an Alloy environment e_{A} such that $\models \Sigma[e_{\mathsf{A}}]$ and $\not\models \varphi[e_{\mathsf{A}}]$. 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 A.7 there exists a proper PDOCFA $\mathfrak{F}$ compatible with e. From Lemma A.5, $\mathfrak{F} \models \{ F(\sigma) : \sigma \in \Sigma \} [e_{\mathsf{P}}]$ and $\mathfrak{F} \not\models F(\varphi)[e_{\mathsf{P}}]$. Then, $\{ F(\sigma) : \sigma \in \Sigma \} \nvdash F(\varphi)$. ∎

3.4 Alloy Integers in PDOCFA

Alloy integers are defined relative to a user-provided bound [Jackson 2006]. This bound, called the *bit width*, is the number of binary digits used to represent integer atoms using 2's complement arithmetic. For instance, with 5 as bit width, we can represent integers -16 through 15. Arithmetic operators are also defined using 2's complement arithmetic. For example, $15 + 1 = -16$ holds when bit width 5 is chosen. We will call $+_{\mathbf{bw}}$ to the arithmetic sum relative to a bit width bw. Numeric atoms may appear in relations like regular atoms do. In fact, they both have the same status.

To support Alloy integers in Dynamite, we enrich PDOCFA theories with new constants, functions, predicates, and their corresponding axioms. We add a new partial identity $iden_{\mathbb{Z}}$, which models the set of Alloy integer atoms in the range $\left[-2^{\mathbf{bw}-1}, 2^{\mathbf{bw}-1} - 1\right]$ determined by the bit width. In our theory, $iden_{\mathbb{Z}}$ is a set of urelements, and $\mathbf{0}$ and $\mathbf{bw}$ are constants that denote the integer value 0 and the bit width, respectively. Notice that while 0 is always contained in $iden_{\mathbb{Z}}$, $\mathbf{bw}$ may not be when $\mathbf{bw} \in \{1, 2\}$. Therefore, we will focus on the general case ($\mathbf{bw} > 2$), and come back to the cases in which $\mathbf{bw} \in \{1, 2\}$ after the presentation of the general case. Axiomatically,

$$iden_{\mathbb{Z}} \text{ in } iden_{\mathsf{U}}, \quad \text{Point}(\mathbf{0}), \quad \text{Point}(\mathbf{bw}), \quad \mathbf{0} \text{ in } iden_{\mathbb{Z}}, \quad \mathbf{bw} \text{ in } iden_{\mathbb{Z}} \ .$$

We introduce a binary predicate symbol $<$ which stands for a linear order with endpoints, over $iden_{\mathbb{Z}}$. In order to simplify the notation, quantifications over $\mathbb{Z}$ are indeed quantifications over points contained in $iden_{\mathbb{Z}}$. We will note by $Max(x)$ the integer unary predicate !some $a : \mathbb{Z} \mid x < a$. A predicate $Min(x)$ is symmetrically defined. Binary predicate $<$ is characterized by the axioms

$$\text{all } a, b : \mathbb{Z} \mid a < b \mid\mid b < a, \quad \text{all } a : \mathbb{Z} \mid !a < a,$$

$$\text{all } a, b, c : \mathbb{Z} \mid (a < b \ \&\& \ b < c) \Rightarrow a < c,$$

$$\text{some } a : \mathbb{Z} \mid Min(a), \quad \text{some } a : \mathbb{Z} \mid Max(a).$$

We introduce next the unary function **succ**, which models the successor function (+1) according to 2's complement arithmetic:

$$\text{all } a, b : \mathbb{Z} \mid (Max(b) \ \&\& \ a < b) \Rightarrow a < \mathbf{succ}(a),$$

$$\text{all } a : \mathbb{Z} \mid !\text{some } b : \mathbb{Z} \mid a < b \ \&\& \ b < \mathbf{succ}(a),$$

$$\text{all } a, b : \mathbb{Z} \mid (Max(a) \ \&\& \ Min(b)) \Rightarrow \mathbf{succ}(a) = b.$$

Having defined successor, defining predecessor, unary minus, addition, subtraction, multiplication, division and power becomes an easy exercise. We present the definition of addition (noted $+_{\mathbf{bw}}$), as an example:

$$\text{all } a : \mathbb{Z} \mid a +_{\mathbf{bw}} \mathbf{0} = a, \quad \text{all } a, b : \mathbb{Z} \mid a +_{\mathbf{bw}} \mathbf{succ}(b) = \mathbf{succ}\left(a +_{\mathbf{bw}} b\right).$$

Once the operations have been defined, we can axiomatize the proper value of the endpoints, as well as remark that **bw** must be greater than 2:

$$Min\left(-2^{\mathbf{bw}-1}\right), \quad Max\left(2^{\mathbf{bw}-1} - 1\right), \quad \mathbf{bw} > \mathbf{succ}(\mathbf{succ}(\mathbf{0}))\ .$$

Supporting the Alloy cardinality operator `#` requires the addition of a new function **card** to PDOCFA. As expected, points have cardinality 1. The cardinality of an arbitrary relation is defined by formulas that, for finite relations, make **card** return the number of tuples[4]:

$$\text{all } r \mid \text{Point}(r) \ \Rightarrow \ \mathbf{card}(r) = \mathbf{succ}(\mathbf{0}),$$

$$\mathbf{card}(\emptyset) = \mathbf{0}, \quad \text{all } r \mid \ \text{Some}(r) \Rightarrow \mathbf{card}(r) = \mathbf{succ}(\mathbf{0}) \ +_{\mathbf{bw}} \mathbf{card}(r \backslash \epsilon(r))\ .$$

Since numeric constants in a specification cannot take values off the range determined by **bw**, for each constant symbol c of type $\mathbb{Z}$ we add axioms:

$$-2^{\mathbf{bw}-1} \leq c, \quad c \leq 2^{\mathbf{bw}-1} - 1\ .$$

Given an Alloy integer expression e, the Alloy expression `Int[e]` denotes the integer atom holding the integer value of e. Conversely, the Alloy function `int` returns the sum of the integer values corresponding to the integer atoms included in a given Alloy expression. For example, `Int[2]` denotes the numeric atom corresponding to the integer 2, which in turn is the result of `int[Int[2]]`.

[4]Unary predicate *Some* characterizes nonempty relations. $\epsilon(A)$ retrieves a pair contained in A, and $\backslash$ stands for set difference.

In PDOCFA we make no distinction between integer values and integer atoms. We will use points contained in $iden_{\mathbb{Z}}$ to represent both kinds on entities. We model function `Int` in PDOCFA with a unary function **Int**, which is defined as the identity. We also introduce a unary function **int** which models `int`. Axioms

$$\text{all } a : \mathbb{Z} \mid \mathbf{int}(a) = a, \quad \text{all } a : univ - \mathbb{Z} \mid \mathbf{int}(a) = \mathbf{0}, \quad \mathbf{int}(\emptyset) = \mathbf{0}$$

state that **int** behaves as the identity on integer atoms, and returns **0** for non integer atoms or the empty relation. For more complex relational expressions, **int** must add the values of the integer points contained in the expression. This is captured by the following axiom:

$$\text{all } r \mid \mathbf{int}(r) = \mathbf{int}(\epsilon(r)) +_{\mathbf{bw}} \mathbf{int}(r \backslash \epsilon(r)).$$

The theories that model Alloy integers in PDOCFA when $\mathbf{bw} \in \{1, 2\}$, are obtained by adequately instantiating the above theory (while at the same time removing axioms $\mathbf{bw}$ in $iden_{\mathbb{Z}}$ and $\mathbf{bw} > \mathbf{succ}(\mathbf{succ}(\mathbf{0}))$). For instance, for $\mathbf{bw} = 1$, the axiomatization of the end points becomes $Min\,(\mathbf{succ}(\mathbf{0}))$ and $Max\,(\mathbf{0})$. Notice that, in this case, $\mathbf{succ}(\mathbf{0})$ is indeed -1.

For PDOCFA models in which the set of integer points is finite, the theory correctly captures Alloy's semantics. Notice that since the theory admits arbitrarily large finite models, by compactness it must admit infinite models as well. We leave the study of such models as further work.

Unlike [Ulbrich et al. 2012], which departs from Alloy's semantics and considers the standard infinite model for integers, we consider 2's complement arithmetic on integers representable using a finite bit width. For Dynamite this is not an optional feature of the language, but rather the only possible choice. In Section 5 we will present the main features of Dynamite. One such feature is the use of the Alloy Analyzer to look for counterexamples of properties being verified. To be useful, counterexamples provided by the Alloy Analyzer must agree with Alloy's semantics as captured in Dynamite's calculus. Otherwise, counterexamples generated by the Alloy Analyzer would fail as counterexamples for the property being verified with the aid of Dynamite.

4. IMPLEMENTATION REMARKS

Implementing Dynamite required solving two tasks, namely,

(1) providing a shallow embedding into PVS of the PDOCFA theories resulting from the translation of Alloy specifications, and
(2) the careful design of the interaction between PVS and the Alloy Analyzer required in order to provide the user with the new commands offered by Dynamite.

The proposed solutions are reported in Sections 4.1 and 4.2, respectively.

4.1 Embedding the Alloy Calculus in PVS

Proving Alloy assertions using Dynamite involves generating a PVS specification. Said specification is obtained as a shallow embedding [Gordon 1989] of the PDOCFA theory resulting from the translation presented in Def. 3.8. PDOCFA theories obtained from Alloy models have in common their logical part (operations, their

meaning and inference rules) presented in Defs. 3.1 and 3.2, while they may differ in the extralogical elements (constants, axioms, theorems) that are directly related to the actual Alloy specification used as input of the translation. Accordingly, the resulting PVS specifications are also composed of two parts, one for handling the logical elements of PDOCFA theories, and another for handling the extralogical ones.

In the general part of the specification the following elements are defined:

—A data type (called `Carrier`) representing the set R of binary relations from Def. 3.1.

—Constants and functions representing the constants and operators from Def. 3.1. For example, the composition operator is represented by the PVS function:

```
composition(x0,x1:  Carrier) :  Carrier
```

—PVS axioms capturing the axioms and inference rules presented in Def. 3.2. For instance,

```
RA_1 :  AXIOM FORALL (x, y, z:  Carrier) :
composition(x,composition(y,z)) = composition(composition(x,y),z).
```

—Auxiliary constants, operators and predicates, as the ones presented in the previous section (π, ρ, $univ_{\mathsf{U}}$, $\otimes$, in, Point), and a few more intended to facilitate the translation process. All of these elements are defined using the elements mentioned in the preceding items. For example, the binary operation $\otimes$ is defined as follows:

```
Cross(x,y: Carrier):Carrier = fork(composition(Pi,x),composition(Rho,y)).
```

PVS natively provides a standard sequent calculus (see [Owre et al. 2001b] for details). The only rule that has to be incorporated is the ω-rule (see Def. 3.2). Using the support for natural numbers offered by PVS, this rule is expressed as a PVS axiom.

We present the translation of the extralogical part of the specification in two steps. We focus first on PDOCFA constants (coming from Alloy signatures and fields) and their properties. We afterwards deal with the translation of functions, predicates, axioms and assertions.

When translating an Alloy signature definition it is necessary to introduce a new symbol for the partial universal relation over atoms from that domain, and another new symbol for the partial identity formed with those atoms. For instance, the translation of signature `Agent` yields the PVS definitions[5]:

```
univ_this?Agent : Carrier  % the partial universal of Agent atoms
iden_this?Agent : Carrier  % the partial identity of Agent atoms
```

In addition, axioms enforcing that these constants have the characteristics mentioned before (being a partial universal relation or a partial identity) must be included. Notice that this part of the translation may generate more axioms depending on the characteristics of the signature being translated (abstract, one sig, extension, etc.).

[5]Notice that in PVS everything at the right of the `%` symbol is considered a comment, and that `?` is a valid character in identifiers.

Restricting quantifiers to range over atoms, as explained in Section 3.2, requires adding for each signature a predicate stating that a relation is a point and it is included in the partial universal relation corresponding to that signature. For signature `Agent`, the predicate is[6]:

```
this?Agent(R: Carrier) : bool =  Point(R) AND Leq(R,univ_this?Agent)
```

When translating field definitions, besides the declaration of the corresponding constant, it is necessary to add appropriate axioms stating the restrictions that the definition imposes on the field. For example, the translation of field `routing` from signature `Domain` leads to the definition of constant `this?Domain?routing` and to the inclusion of the following axiom[7]:

```
this?Domain?routing : AXIOM FORALL (this: (this?Domain)) :
   Leq( this?Domain?routing,
        CartesianProduct( univ_this?Domain,
                          CartesianProduct( Navigation(this,this?Domain?space),
                                            Navigation(this,this?Domain?endpoints))))
```

This axiom establishes that `this?Domain?routing` denotes a relation in which, for each tuple $\langle d, i \star g \rangle$, address i is in the space of domain d, and agent g is an endpoint for d.

The translation of predicates, functions, facts and assertions is direct. It is sufficient to translate the formula (or expression) that defines each of these constructs and add the corresponding predicate, function, theorem or axiom to the resulting PVS specification. For example, assertion `BindingPreservesReachability` (shown in Fig. 4) is translated to the PVS theorem:

```
BindingPreservesReachability : THEOREM
FORALL (d: (this?Domain2), d??_: (this?Domain2),
  newBinding: Carrier |
      Leq( newBinding,CartesianProduct(univ_this?Identifier,univ_this?Identifier))):
    this?IdentifiersUnused(d,Navigation(newBinding,univ_this?Identifier))
    AND this?AddBinding(d,d??_,newBinding)
      IMPLIES FORALL (i: (this?Identifier), g: (this?Agent)) :
        this?ReachableInDomain(d,i,g) IMPLIES this?ReachableInDomain(d??_,i,g)
```

On top of this notation pretty-printing algorithms are applied to PVS formulas occurring during the development of proofs. Therefore, the user only sees Alloy syntax while working within Dynamite. This is one of the important features of Dynamite because it makes it unnecessary for Alloy users to learn another formalism in order to prove the given assertions.

Embedding Alloy integers. The characterization of Alloy integers in PDOCFA presented in Section 3.4 can be easily embedded in PVS as a new PVS theory. Such an embedding would be suboptimal, since it would miss all the support provided by PVS for reasoning about integer arithmetic. We will instead use a new PVS theory `fint` (for *finite ints*), parameterized by the bit width. This theory profusely uses theory `int` provided by PVS. For example:

[6] `Leq` is the predicate corresponding to the set inclusion operator *in*.
[7] `Navigation` is the operator corresponding to •, and `CartesianProduct` simulates the behaviour of Cartesian product between relations of arbitrary arity.

—the bit width (noted as `bitwidth`), is a formal parameter of the theory and has type `posnat` (i.e., positive natural).

—the minimum and the maximum of the interval determined by the bit width are modeled by the integer constants `min_fint` and `max_fint` defined as

$$-\texttt{exp2}(\texttt{bitwidth} - 1) \quad \text{and} \quad \texttt{exp2}(\texttt{bitwidth} - 1) - 1,$$

respectively.

—a PVS predicate is defined for delimiting the numbers in this interval:

```
inRange_fint(n: int): bool = min_fint <= n and n <= max_fint
```

—this PVS theory includes the definition of a subtype of `int`, called `fint`, that represents the Alloy integers in the interval:

```
fint: type = { n: int |  inRange_fint(n) }
```

—all the integer operations supported in Alloy (addition, subtraction, multiplication, integer division and remainder) are modeled as PVS functions on `fint`. For example, addition is defined as

```
add_fint(n1, n2: fint): fint =
   if inRange_fint(n1+n2) then n1+n2
   elsif n1+n2 > max_fint
      then (min_fint-1) + (n1+n2) - max_fint
      else max_fint + (n1+n2) - (min_fint-1)
   endif
```

As discussed at the end of section 3.4, it must be noted that this theory does not support models in which the set of integer atoms is not finite.

4.2 Overview of Dynamite's Architecture

The current prototype of the Dynamite Proving System was developed as an extension of PVS. Therefore, we wrote Emacs extensions (for system commands such as those for opening and editing an Alloy specification), Lisp routines that interact with the PVS prover engine (to implement the Dynamite-specific commands, the pretty-printing of the formulas, etc.) and Java code (whose purpose is the translation and validation of formulas, goals and specifications, as well as the postulation of witness candidates for existentially quantified assertions, among others).

A component-and-connector view diagram of Dynamite's architecture showing the interactions between the main components of the system[8] is depicted in Fig. 3.

As explained in [Owre 2008], the PVS prover engine runs as a subprocess of Emacs, trough an ad-hoc ILISP interface [Kaufmann et al. 2002]. We added the implementation of the Dynamite-specific commands explained in previous sections

[8]It is worth noting that, as usual in C&C diagrams, despite being of the same type, not all the client-server connectors showed in the figure are implemented in the same way. For example, the connectors between the "Dynamite Translator" and the Alloy Analyzer are implemented using the API exposed by the latter, while the connectors linking the "Dynamite proof commands processor" and the "Dynamite Translator" are implemented through the OS standard input/output subsystem.

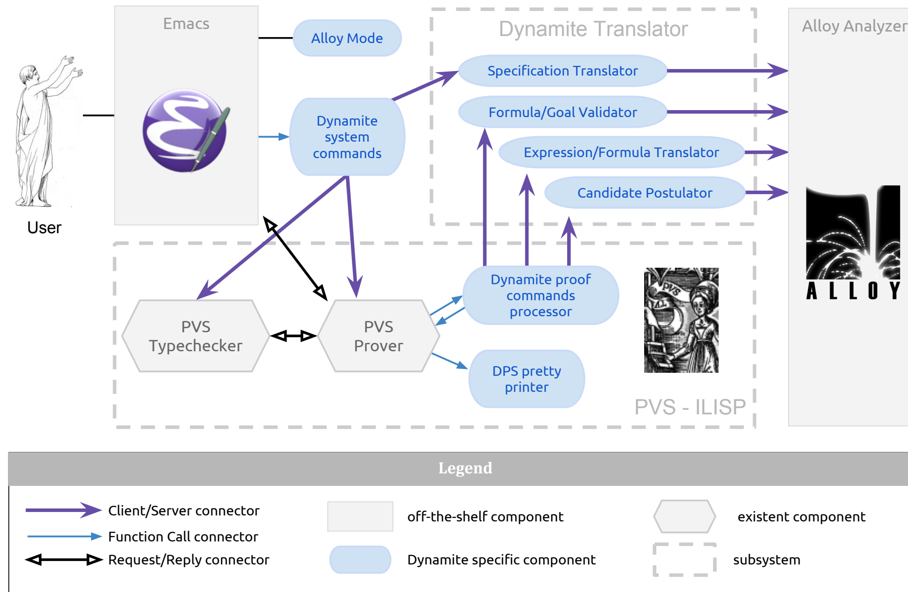


Fig. 3. C&C view of the Dynamite architecture.

to this engine, as PVS strategies and rules ("Dynamite proof commands processor" in the diagram). These extensions are conservatively sound with respect to the logic of PVS.

We also modified the PVS function for pretty-printing in order to allow the user to see Alloy formulas in the sequents as long as it can be done. At any point during a proof the user can deactivate and activate (when the translation back is possible) the pretty-printer.

The "Dynamite proof commands processor" is responsible for the interaction with the Java processes that, using the Alloy Analyzer, validate formulas and goals during the proof, suggest the elimination of (presumably) unnecessary formulas from the sequent, and postulate expressions that can be used to instantiate existential quantifiers. These Java processes are collectively referred to as "Dynamite Translator"[9].

As Dynamite is an extension of PVS, all the regular PVS proof commands are available to the user. Some of them, such as `case` and `inst`, take formulas or expressions as parameters. When the pretty-printer is activated, the user can write Alloy formulas and expressions as parameters for these commands. The "Dynamite proof commands processor" is also responsible for the translation of those parameters to the corresponding PDOCFA formulas and expressions, via the "Dynamite Translator".

Dynamite includes Emacs extensions that allow the user to open an Alloy specification, generate the corresponding PDOCFA theories and start, or redo, the proof

[9]Besides inaccuracy, the name is maintained for historical reasons.

of any of its assertions (among other system-level functionalities). All these functions can be accessed trough the user menu. Additionally, Dynamite has an Emacs major mode ("Alloy mode") that provides syntax highlighting for the manipulation of Alloy code.

5. FEATURES OF DYNAMITE

Proving properties can be seen as a handcraft discipline. It usually requires a high level of training on the methods adopted to develop the proofs, a deep understanding of the concepts formalized in the theory and, most of the time, lots of patience. Even more so if we consider that the person in charge of proving the correctness of the assertions is (many times) not the person who wrote the model.

Automatic verification of proofs is somewhat comparable to the spell checker in text editors. It does not help you compose a text, it just guarantees the absence of syntactic mistakes. To be considered useful, an interactive theorem prover must be capable of helping the user with the proving process.

Dynamite is more than a proof checker, and in this article we report our experience in proving properties of a complex model. The key ingredient that makes Dynamite much more than syntactic sugar on top of PVS is the use of the Alloy Analyzer as a helper during the development of proofs. The most important cases in which Dynamite ends up being particularly helpful occur when the user:

—introduces a new hypothesis or a new lemma,

—needs to separate a proof in cases,

—wants to hide some formulas,

—wishes to prune some of the formulas presented by the theorem prover in the subgoals produced by the application of a rule,

—has to prove an existentially quantified formula.

When a user starts a proof, the proof tree has only one node. This tree will grow as the result of the application of *proof rules*. These applications may result in one or more nodes that are direct ascendants of the node to which the proof rule was applied. Consider for instance the following proof rules for conjunction:

$$\frac{\alpha, \beta, \Gamma \vdash \Delta}{\alpha \wedge \beta, \Gamma \vdash \Delta} \wedge\vdash \qquad\qquad \frac{\Gamma \vdash \Delta, \alpha \qquad \Gamma \vdash \Delta, \beta}{\Gamma \vdash \alpha \wedge \beta, \Delta} \vdash\wedge .$$

Rule $\wedge\vdash$ shows that proving a sequent that contains the conjunction $\alpha \wedge \beta$ in the antecedent reduces to proving a sequent with both α and β in the antecedent (in this case, a single new sequent has to be proved). Similarly, rule $\vdash\wedge$ shows that proving a sequent with $\alpha \wedge \beta$ in the consequent reduces to proving two different sequents: one with α in the consequent, and another one with β in the consequent.

In the following sections we will show the usefulness in the development of proofs of each of the features contributed by Dynamite. We will use as a running example an Alloy model for binding in network domains presented in [Zave 2006].

5.1 A Running Example: Binding in Network Domains

The model, presented in [Zave 2006], deals with the formal definition of a mechanism for binding of identifiers in the delivery of messages in computer networks. Thus, the

model is not a specification of an isolated software or hardware artifact, but rather the specification of network services whose implementation may involve several software and hardware *agents*. The model describes how communicating agent identifiers are bound so that the messages reach their correct destination. Properties about the possibility of reaching an agent, determinism in the delivery of messages, existence of cycles in the routing of messages and the possibility of constructing a return path for a message are formally specified in the model. In particular, the model studies how these properties are affected by the addition of new bindings between identifiers.

When an agent wants to send a message to another agent a *communication* is established. That communication may involve intermediary agents that just forward the message in its way to its destination. The original sender of the message and its intended final receiver are called the *endpoints* of the communication. Endpoints are organized into *domains*. Each domain has its own set of endpoints, and uses identifiers to recognize them. Identifiers are called *addresses*. Additionally, a domain keeps track of how agents are identified by particular addresses. *Paths* describe connections from a *generator* agent to an *absorber* agent assuming the generator can be recognized by address *source* and the absorber by address *dest*.

A simplified version of the previous concepts in Alloy takes the following form:

```
sig Agent {}
abstract sig Identifier { }
sig Address extends Identifier { }

sig Domain {
    endpoints: set Agent,
    space: set Address,
    routing: space -> endpoints
}

sig Path {
    source: Address,
    dest: Address,
    generator: Agent,
    absorber: Agent
}
```

A domain *supports* a path if the connections described by the path are consistent with the domain. The following predicate characterizes when a domain supports a path:

```
pred DomainSupportsPath [d: Domain, p: Path] {
   p.source in (d.routing).(p.generator) and
   p.absorber in (p.dest).(d.routing)
}
```

When a message has to be send to an endpoint, the identifier used by the initiator to indicate the destination must be bound to the identifier used by the domain to locate the receiver. This binding is done in three ways. The simplest one is when the initiator is responsible for performing the binding. The message sent has as destination the actual identifier of the receiver. The second scenario occurs when the message sent by the initiator is delivered to an agent that is not the intended receiver. This agent, called *handler*, looks up the corresponding binding, updates the destination address and forwards the message. The third one is basically the same as the second one, but the original destination identifier is composed of two

parts which are used by the handler to locate the next agent in the forwarding chain. These communication patterns show the need for some distinction in the identifiers used in the model. Besides addresses, there will be unrestricted identifiers called *names*, and complex identifiers used in the third kind of communications. Thus, signature `Identifier` is extended by signatures `Name` (modeling unrestricted identifiers) and `AddressPair` (for compound identifiers):

```
sig Name extends Identifier { }
sig AddressPair extends Identifier {
   addr: Address,
   name: Name
}
```

The binary relations `addr` and `name` are defined to formalize the structure of complex identifiers.

The possible bindings in each domain are specified by a ternary relation

$$\texttt{dstBinding} \subseteq \texttt{Domain} \times \texttt{Identifier} \times \texttt{Identifier}\ .$$

We introduce `dstBinding` (destination binding) by extending the signature `Domain`, and constraining its meaning with a signature axiom.

```
sig Domain2 extends Domain {
   dstBinding: Identifier -> Identifier
} {
  all i: Identifier | i in dstBinding.Identifier implies
   (
     (i in Address implies i in space) and
     (i in AddressPair implies i.addr in space)
   )
}
```

Paths are also extended in order to include a new field `origDst` representing the identifier originally given as destination:

```
sig Path2 extends Path {
   origDst: Identifier
}
```

A predicate `AddBinding` states how a domain is affected by the addition of new bindings:

```
pred AddBinding[d,d':Domain2, newBinding:Identifier -> Identifier] {
-- Precondition: the new bindings can be applied in the domain.
   all i: Identifier | i in newBinding.Identifier implies
   (  (i in Address implies i in d.space) and
      (i in AddressPair implies i.addr in d.space)
   ) and
-- Postconditions:
   d'.endpoints = d.endpoints and
   d'.space = d.space and
```

```
   d'.routing = d.routing and
   d'.dstBinding = d.dstBinding + newBinding
}
```

An agent g is considered "*reachable* in a domain d from an identifier i" if:

—i is connected to an address a in the reflexive–transitive closure of the binary relation formed by all the bindings corresponding to d,

—a cannot be bound to another identifier in d, and

—in domain d, a can route messages to g.

Following Zave's model, and recalling that "*" denotes reflexive–transitive closure in Alloy, reachability is modeled by the predicate

```
pred ReachableInDomain [d: Domain2, i: Identifier, g: Agent] {
   some a: Address |
      a in i.*(d.dstBinding) and
      a !in (d.dstBinding).Identifier and
      g in a.(d.routing)
}
```

Figure 4 presents assertion `BindingPreservesReachability`. This assertion states that if an agent is reachable in a domain d, it is also reachable in the domain resulting from adding a new binding to d, provided that the newly bound identifiers are not used in d. This latter condition is formalized by the following predicate `IdentifiersUnused`:

```
pred IdentifiersUnused [d: Domain2, new: Identifier ] {
   no ((d.routing).Agent & new) and
   no ((d.dstBinding).Identifier & new) and
   no (Identifier.(d.dstBinding) & new)
}
```

```
assert BindingPreservesReachability {
   all d,d': Domain, newBinding: Identifier->Identifier |
      IdentifiersUnused[d,newBinding.Identifier] and
      AddBinding[d,d',newBinding]
        implies (all i: Identifier, g: Agent |
                 ReachableInDomain[d,i,g] implies ReachableInDomain[d',i,g])
}

check BindingPreservesReachability for 4 but 2 Domain
```

Fig. 4. A nontrivial assertion: `BindingPreservesReachability`.

A domain is called *deterministic* if each identifier is associated to at most one agent. Assertion `BindingPreservesDeterminism` states that

> *whenever a new binding for an unused identifier is added to a deterministic domain, it remains deterministic.*

A domain is considered *non-looping* if the transitive closure (denoted by ˆ) of the bindings for that domain has no cycles. Assertion `BindingPreservesNonlooping` then states that

> *the addition of a new binding to a non-looping domain keeps this condition as long as the transitive closure of the new binding does not have cycles.*

Another desirable property of a network is the capability to send a message to the sender of a previously received message. This is called *returnability*. A domain in which it is possible to return the received messages is called a *returnable* domain. In order to write conditions ensuring returnability of a domain, it is necessary to study how the source identifiers can be modified by the handlers that forward the message, because the final source identifier is used by the receiver as the destination of the return message. An assertion `StructureSufficientForReturnability` is also modeled in [Zave 2006].

In [Zave 2006], Zave used the Alloy Analyzer to analyze the model and concluded that the previously presented assertions hold for Alloy domains containing at most 2 network domains and 4 elements in each set (such as identifiers, agents, etc). Using Dynamite we proved that all these assertions hold independently of the maximum amount of elements in each set.

5.2 Introduction of Hypotheses and Lemmas

Many times, when attempting to prove a sequent $\Gamma \vdash \Delta$, a new hypothesis α is introduced as a means to simplify and modularize the proof. Hypothesis α can then be used in the proof of the sequent, but will have to be discharged later. A rule as the one just described can be implemented using the PVS rule case. Still, we may want to go a step further and gain some confidence on the suitability of formula α. Does α actually follow from Γ?. It is frustrating to realize, after finishing the proof with the aid of formula α, that the new hypothesis cannot be discharged, deeming the previous proof effort useless. In order to reduce the risk of introducing inappropriate hypotheses, Dynamite introduces the rule dps-hyp:

$$\frac{\Gamma, \alpha \vdash \Delta \qquad \Gamma \vdash \alpha}{\Gamma \vdash \Delta} \ \textsf{dps-hyp}(\alpha) \ .$$

The use of rule dps-hyp triggers a call to the Alloy Analyzer in order to analyze whether sequent $\Gamma \vdash \alpha$ follows from the model. If a counterexample is found within the provided scopes, it is reported to the user and the hypothesis is removed. Let us see the schematic representation of the proof tree for assertion `BindingPreservesReachability` (one of the properties we proved), shown in Fig. 5. A dps-hyp command was applied in each grey node. It is worth noting that those nodes are the main reason why a branch splitting occurs in that example. This shows that a mistake in the introduction of a case can invalidate a major part of the proof.

A similar situation occurs when a lemma is introduced along a proof as a means to modularize the proof effort. Proof rule dps-lemma calls the Alloy Analyzer in order to analyze whether the introduced lemma is indeed valid.

The experience in using Dynamite on the case study presented here showed us

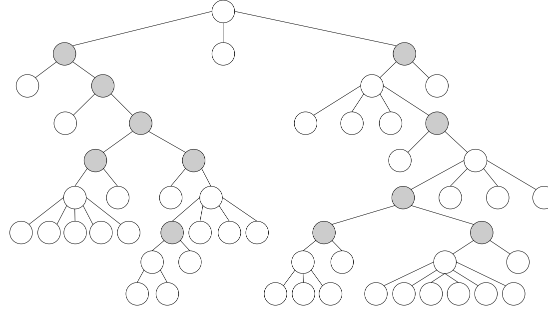


Fig. 5. Simplified proof tree for the assertion `BindingPreservesReachability`.

that this feature is a dramatic improvement with respect to the standard case introduction. If a counterexample is found it is shown to the user, so the hypothesis or lemma can be corrected using the information revealed by the counterexample. Counterexamples also give the user a better grasp on the model because they expose tricky corner cases.

5.3 Introduction of cases

The application of the Dynamite dps-case command splits the current branch into two branches, using a provided formula α as a parameter. In one of the branches α appears as a new formula in the antecedent, and it is placed in the consequent in the other branch:

$$\frac{\Gamma, \alpha \vdash \Delta \qquad \Gamma \vdash \Delta, \alpha}{\Gamma \vdash \Delta} \ \mathsf{dps\text{-}case}(\alpha) \ .$$

Notice that having α as a proof obligation is equivalent to having $\neg\alpha$ as a hypothesis, which explains why α allows us to separate the proof into cases. The Dynamite dps-case command improves over the regular PVS case command by using the Alloy Analyzer in order to automatically search for models of the formulas

$$\left(\bigwedge_{\gamma \in \Gamma} \gamma \right) \wedge \alpha, \qquad \text{and} \qquad \left(\bigwedge_{\gamma \in \Gamma} \gamma \right) \wedge \neg\alpha \ .$$

The existence of the models guarantees that formula α indeed splits into meaningful cases. If the Alloy Analyzer does not yield a model for any of the formulas, this is reported to the user.

5.4 Hiding sequent formulas

During the development of a proof the amount of hypotheses tends to grow. For example, new hypotheses are introduced when a case splitting is performed. The information expressed in these hypotheses may be useful for closing some branches and useless for some others. Thus, a sequent may contain formulas that are irrelevant to close branches originating in the sequent. For example, after a branch splitting some formulas may no longer be needed for some of the sub-goals, and be necessary to prove others.

In Fig. 6 we show an open branch that was obtained during the proof of property `BindingPreservesReachability`, reached after a few applications of proof commands. Notice that, even when the only relevant formulas of the sequent are `1` and `-1`, the other 9 formulas in the sequent obfuscate the job of proving the assertion, turning the sequent very difficult to understand at first glance. This is a situation that occurs quite often. For instance, predicates are often used to wrap several related concepts which apply in different sub-goals. Using one of those concepts requires us to expand the predicate. Doing this will not only result in the appearance of the desired formula as hypothesis, but the rest of the sub-formulas will also appear as part of the sequent.

```
{-1}  (a!1 in (i!1 . (*(d!1 . dstBinding!1))))
[-2]  (IdentifiersUnused (d!1, (newBinding!1 . Identifier)))
[-3]  (all i : Identifier| ((i in (newBinding!1 . Identifier)) =>
      (((i in Address) => (i in (d!1 . space!1))) &&
      ((i in AddressPair) => ((i . addr!1) in (d!1 . space!1))))))
[-4]  ((d'!1 . endpoints!1) = (d!1 . endpoints!1))
[-5]  ((d'!1 . space!1) = (d!1 . space!1))
[-6]  ((d'!1 . routing!1) = (d!1 . routing!1))
[-7]  ((d'!1 . dstBinding!1) = ((d!1 . dstBinding!1) + newBinding!1))
[-8]  (g!1 in (a!1 . (d!1 . routing!1)))
 |-------
{1}   (a!1 in (i!1 . (*((d!1 . dstBinding!1) + newBinding!1))))
[2]   (a!1 in (i!1 . (*(d'!1 . dstBinding!1))))
[3]   (a!1 in ((d!1 . dstBinding!1) . Identifier))
```

Fig. 6. Example of a sequent obtained after the application of several proof commands.

To solve this, it is common for interactive theorem provers to provide commands for hiding formulas from a goal, under the assumption that they will not be used. On the other hand, the use of this command also presents a risk. If, by mistake, a relevant formula is hidden, the user will not be able to close the branch. Given a sequent $\Gamma \vdash \Delta$ (with $\Gamma = \{ \gamma_1, \ldots, \gamma_k \}$ and $\Delta = \{ \delta_1, \ldots, \delta_m \}$) result of hiding some formulas, the Dynamite command dps-validate-goal automatically searches for counterexamples of the logical implication between the conjunction of the formulas in the antecedent and the disjunction of the formulas in the consequent:

$$\left(\bigwedge_{1 \leq i \leq k} \gamma_i \right) \Rightarrow \left(\bigvee_{1 \leq j \leq n} \delta_j \right) . \quad (3)$$

In this way, if a counterexample is found, it means that the proof objective cannot be reached because the hypotheses are not sufficient to prove the desired property. If that goal is the result of hiding some formulas from a sequent for which a similar analysis did not return a counterexample, it means that some of the newly hidden formulas were necessary.

5.5 Pruning of goals

As we explained in Section 5.4, sequents can grow up to a point in which they get very difficult to be understood. A handy and time-saving feature is the use of the Alloy Analyzer that Dynamite does to prune goals.

Let us assume we are proving a sequent $\Gamma \vdash \Delta$ (where $\Gamma = \{ \gamma_1, \ldots, \gamma_k \}$ and $\Delta = \{ \delta_1, \ldots, \delta_m \}$) from a theory $\Omega = \{ \omega_1, \ldots, \omega_n \}$. In order to reduce the proof search space we will try to identify formulas from Γ, Δ and Ω that can be safely removed. Notice that having fewer formulas actually reduces the proof search space. Many proof attempts that could depend on the removed formulas (rules for instantiation, rewriting, or applying strategies) are now avoided. This reduces the number of instantiations of inference rules that the theorem prover has to consider, as well as helps the user stay focused on the relevant parts of the sequent.

When the Dynamite proof command dps-hide is applied on sequent $\Gamma \vdash \Delta$, the system builds an Alloy model containing the original Alloy model Ω under analysis and an assertion on the validity of formula (3). Notice that analyzing with the Alloy Analyzer the newly built model Ω' will not return counterexamples (otherwise sequent $\Gamma \vdash \Delta$ would not be valid). We then request the Alloy Analyzer for an UnSAT-core of Ω'. An Alloy UnSAT-core [Torlak et al. 2008] of Ω' is a subset of Ω' that is also inconsistent (notice that inconsistency is defined up-to the considered scopes). The UnSAT-cores retrieved by the Alloy Analyzer do not need to be minimal, but many times are proper subsets of the premises and consequents of the sequent and theory under analysis. Command dps-hide then hides all those formulas in Ω, Γ and Δ that are not part of the retrieved UnSAT-core. As in all the previous commands that involve using the Alloy Analyzer, it may be the case that a necessary formula is hidden. In our case-study, out of the 69 times command dps-hide was used, only in one occasion a necessary formula was hidden. But, equally important, the command allowed us to identify two places where unnecessary detours (proof steps involving formulas that could be removed) were taken. Removing detours allowed us to reduce the proof lengths from 969 proof steps to 573.

This feature was one of the most useful in the verification of the model presented in this paper. The importance of this rule is that it provides a guide in the construction of the proof by revealing those formulas that will be needed to prove the property. For instance, the relevant formulas in the sequent depicted in Fig. 6 (formulas -1 and 1) are automatically identified by applying this command.

5.6 Automated Witness Generation

When proving the property `BindingPreservesDeterminism`, the sequent depicted in Fig. 7 is produced. Notice that the formula in the consequent is existentially quantified. According to the proof calculus for PVS described in [Owre et al. 2001b], in order to prove the sequent we must find a suitable witness (i.e., a term that, when substituted for variable `ai`, makes the resulting sequent provable):

$$\frac{\Gamma \vdash A\{x \leftarrow t\}, \Delta}{\Gamma \vdash (\exists x : A), \Delta} \vdash \exists$$

A closer look at the proof calculus shows that the only rules that require creativity are the Cut rule [Owre et al. 2001b, p. 17] (where an appropriate formula A has

```
(-1) (a in (i . ^ ((newBinding + (d.dstBinding)))))
(-2) no (((Identifier . (d.dstBinding)) & (newBinding . Identifier)))
(-3) no ((((d.dstBinding) . Identifier) & (newBinding . Identifier)))
(-4) (not (a in (i . ^ newBinding)))
(-5) (not (a in (i . ^ (d.dstBinding))))
|---
(1) some ai: Identifier | ai in i.^newBinding && a in ai.^(d.dstBinding)
```

Fig. 7. Sequent with existentially quantified conclusion.

to be determined), the rules for quantifiers [Owre et al. 2001b, p. 18], or rules for equational reasoning where algebraic properties of the relational operators are to be applied. Notice that rule $\forall \vdash$ can be substituted (by promoting the universally quantified formula from the antecedent to the consequent of the sequent) by an application of rule $\vdash \exists$. In case we are confronted with a sequent with either a universally quantified formula in the consequent, or an existentially quantified formula in the antecedent, the quantifiers can be skolemized away through the application of proof rules $\vdash \forall$ and $\exists \vdash$, respectively. The Cut-elimination theorem [Gentzen 1935] states that proofs can be replaced by (usually more complex) proofs that do not use the Cut rule. In [Frias 2002, Thm. 5.2] it is proved that PDOCFA terms that do not include reflexive-transitive closure can be translated to equivalent first-order logic formulas. Therefore, creativity in proofs can be pinpointed to those places where:

(1) equational reasoning involving reflexive-transitive closure is required, or
(2) witnesses for the application of rule $\vdash \exists$ must be produced.

In order to reduce user intervention, in this section we will present an effective technique that uses the Alloy Analyzer in order to automatically generate witness candidates. Also, we will present several examples where the application of the proposed technique yields the required witnesses.

In Alg. 1 we present the algorithm we use for witness candidate generation. Recall that *environments* are the semantic structures in which Alloy models are evaluated.

In the following paragraphs we will explain the algorithm and argue about its correctness, as well as discuss in what conditions the algorithm may fail to produce a candidate. Afterwards we will describe several experiments we performed.

5.6.1 *The Inputs to the Algorithm (line 1).* Given a sequent $\Gamma \vdash \texttt{some x} : \texttt{T} \mid \alpha(\texttt{x})$ that has to be proved, the algorithm receives as inputs the set Γ and the formula $\texttt{some x} : \texttt{T} \mid \alpha(\texttt{x})$ for which the witness must be produced.

5.6.2 *Initialization (lines 2–5).* Variable *DW* will store those witness precandidates that are eventually discarded. Variable *result* stores the output of the algorithm, and its content will be discussed in Section 5.6.3. Variable $\mathcal{E}$ is initialized with $\mathcal{E}_0$, any environment in which Γ holds. $\mathcal{E}_0$ is produced by invoking the Alloy Analyzer.

5.6.3 *The Output (variable result).* Variable *result* returns a *coverage* for the formula under analysis. A coverage is a set of terms $\{ t_1, \ldots, t_k \}$ such that the Alloy Analyzer is able to verify the sequent $\Gamma \vdash \alpha(t_1) \mid\mid \cdots \mid\mid \alpha(t_k)$. The following

```
1  witnessCandidate(Γ, δ)                                  /* δ has the form some x : α[x] */
2  │ DW ← ∅;            /* DW will store the discarded witnesses found so far */
3  │ result ← ∅;
4  │ ℰ ← ℰ0;                          /* ℰ0 is an environment such that ℰ0 ⊨ Γ */
5  │ α′ ← α;
6  │ while (result == ∅ ∧ ℰ != null) do
7  │ │ if there exists a witness precandidate t in ℰ then
8  │ │ │ if t is a valid witness precandidate then
9  │ │ │ │ result ← result ∪ { t };
10 │ │ │ else
11 │ │ │ │ DW ← DW ∪ { t };
12 │ │ │ │ ℰ ← ℰi;                  /* ℰi is the environment in which t failed */
13 │ │ │ │ for each t′ ∈ shrunkenWitnessesFrom(t) do /* shrunkenWitnessesFrom(t)
   │ │ │ │ are the witnesses with the same syntactic structure as t but with
   │ │ │ │ every constant c replaced by a constant denoting a subset of c */
14 │ │ │ │ │ if t′ is a valid witness precandidate then
15 │ │ │ │ │ │ result ← result ∪ { t′ };
16 │ │ │ │ │ end
17 │ │ │ │ end
18 │ │ │ │ if (result == ∅) then
19 │ │ │ │ │ if in every environment t contains atom a such that α′[a] holds then
20 │ │ │ │ │ │ α′ ← α′[t & x];                  /* α′ has been relativized */
21 │ │ │ │ │ else
22 │ │ │ │ │ │ if there is a coverage C ⊆ DW then
23 │ │ │ │ │ │ │ result ← C;
24 │ │ │ │ │ │ end
25 │ │ │ │ │ end
26 │ │ │ │ end
27 │ │ │ end
28 │ │ else
29 │ │ │ if α′ has been relativized then
30 │ │ │ │ α′ ← α;
31 │ │ │ else
32 │ │ │ │ ℰ ← null;
33 │ │ │ end
34 │ │ end
35 │ end
36 end
```

Algorithm 1: Algorithm for witness candidate generation.

simple Alloy model shows that coverages are many times necessary.

```
sig A {}
sig B in A {}
one sig x1, x2 in A {}
fact { x1 in B || x2 in B }
assert needsCoverage { some x : A | x in B }
```

In Alloy notation, signature `B` denotes a subset of `A`, and objects `x1` and `x2` belong to `A` (and since `B` is contained in `A`, also perhaps to `B`). Notice that the fact guarantees that the assertion is indeed valid. Yet no single witness exists. In some environments `x1` will be a witness, and in others the witness will be `x2`. Notice also that:

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \alpha(t_1), \ldots, \alpha(t_n) \vdash \alpha(t_1), \ldots, \alpha(t_n)}\,(\mathrm{Ax})}{\Gamma, \alpha(t_1) \mid\mid \cdots \mid\mid \alpha(t_n) \vdash \alpha(t_1), \ldots, \alpha(t_n)}\,(\mid\mid \vdash \times\, n) \qquad \dfrac{\dfrac{}{\Gamma \vdash \alpha(t_1) \mid\mid \cdots \mid\mid \alpha(t_n)}\,(\mathrm{Coverage})}{\Gamma \vdash \alpha(t_1) \mid\mid \cdots \mid\mid \alpha(t_n), \alpha(t_1), \ldots, \alpha(t_n)}\,(\mathrm{Weak})}{\Gamma \;\vdash\; \alpha(t_1), \ldots, \alpha(t_n)}\,(\mathrm{Cut})}{\Gamma \;\vdash\; \texttt{some x : T} \mid \alpha, \ldots, \texttt{some x : T} \mid \alpha}\,(\vdash \exists \times n)
$$
$$
\dfrac{\Gamma \;\vdash\; \texttt{some x : T} \mid \alpha, \ldots, \texttt{some x : T} \mid \alpha}{\Gamma \;\vdash\; \texttt{some x : T} \mid \alpha}\,(\mathrm{Contraction} \times n)
$$

Fig. 8. Use of coverage $\{ t_1, \ldots, t_n \}$ for proving an existential formula.

```
sig A {}
one sig cA extends A {}
sig B {
   f1 : A
}
assert exstentialAssert { some x : B | alpha(x)}
```

Fig. 9. A Sample Alloy Model.

—`x1 in B || x2 in B` holds as per the fact, and

—`(x1 in B || x2 in B) => some x:A | x in B` holds.

Therefore, the coverage $\{ \texttt{x1}, \texttt{x2} \}$ allows us to prove the existential formula. This reasoning is easily generalized. The proof-schema in Fig. 8 shows that a coverage allows us to prove the existentially quantified formula.

5.6.4 *Building a Witness Precandidate (line 7).* This is one of the main contributions of Section 5.6. The precandidate is built by internalizing Alloy's syntax inside an Alloy model. We will present the technique by means of a simple running example. Let us consider the Alloy model presented in Fig. 9. The model is instrumented with appropriate signatures, functions and predicates. In Fig. 10 we present a fragment of the resulting Alloy model.

The instrumented model introduces new signatures that model syntactic internalizations of the source model signatures and fields, as well as of the relational operators (in Fig. 10 we only include the union of unary relations and the intersection of binary relations). We also include a number of facts the preclude redundant instances. For example, fact `UnarySumOperand0IsNotUniv` states that the first operand in a sum cannot be the universal relation (after all, the result of the union would be the set `univ`). Several other properties of this kind are included.

Finally, using the Alloy Analyzer we look for an environment that satisfies formula `witnessSearch`. The environment allows us to retrieve a term `t` that denotes a nonempty set in which all objects satisfy formula `alpha`. This is the witness precandidate. In order to prevent the analysis from returning previously discarded terms, the model includes a fact that is iteratively enriched in order to prevent previously discarded terms from being produced.

5.6.5 *Validating a Witness Precandidate (lines 8–9).* A witness precandidate `t` is valid in an environment. In contrast, a witness candidate must be valid in *all* environments. In order to analyze whether `t` can be considered as a witness candidate, we modify the Alloy model from Fig. 9 by replacing assert `existentialAssert` by

```
sig A {}
one sig cA extends A {}
sig B { f : A }

abstract sig Term {
  complexity : Int
}
abstract sig UnaryTerm extends Term {
  unaryValue : set univ
}
abstract sig BinaryTerm extends Term {
  binaryValue : univ -> univ
}
one sig UnivSyntax extends UnaryTerm{}{ unaryValue = univ }
one sig A_Syntax extends UnaryTerm{}{ unaryValue = A }
one sig cA_Syntax extends UnaryTerm{}{ unaryValue = cA }
one sig B_Syntax extends UnaryTerm{}{ unaryValue = B }
one sig f_Syntax extends BinaryTerm{}{ binaryValue = f }

fact sigComplexity{A_Syntax.complexity=1 and cA_Syntax.complexity=1 and
                   B_Syntax.complexity=1 and f_Syntax.complexity=2}
sig UnarySum extends UnaryTerm {
  operand0, operand1 : UnaryTerm
}{ complexity = operand0.complexty + operand1.complexity + 1
   unaryValue = operand0.unaryValue + operand1.unaryValue }
sig BinaryIntersection extends BinaryTerm {
  operand0, operand1 : BinaryTerm
}{ complexity = operand0.complexty + operand1.complexity + 1
   binaryValue = operand0.binaryValue & operand1.binaryValue }

fact UnarySumOperand0IsNotUniv { (all t : UnarySum | t . operand0 !in UnivSyntax)) }

run witnessSearch { some t : UnaryTerm |
                      some t.unaryValue and all v : t.unaryValue | alpha(v) }
```

Fig. 10. Fragment of Alloy model with internalized Alloy syntax.

the following:

```
assert witnessPrecandidateValidation {
                          some t and all v : t | alpha(v) }
```

The assertion requires `t` to denote a nonempty set in which all the elements satisfy formula `alpha` *in all environments*. If no counterexamples are produced by the Alloy Analyzer, term `t` is stored in the algorithm output variable *result* and promoted to witness candidate.

5.6.6 *Witness Precandidate Failure by Over-Approximation (lines 13–20).* Let us assume that witness precandidate `t` fails in environment $\mathcal{E}_i$. According to Section 5.6.5, this implies that either `t` is empty in $\mathcal{E}_i$ or some value from the set denoted by term `t` does not satisfy `alpha`. The second condition may hold even if `t` provides at least one value that satisfies `alpha` in each environment. In this case we say that term `t` *over-approximates* a witness candidate. In order to avoid this over approximation we will use two techniques:

(1) Recalling that Alloy signatures are constants that may appear in term t, term t may fail to be a witness candidate because it includes a signature that is larger than necessary. For instance, in [Zave 2006] we have as part of the signature hierarchy

```
sig Domain3  extends  sig Domain2  extends  sig Domain .
```

Semantically, the sets denoted by the signatures satisfy

$$Domain3 \subseteq Domain2 \subseteq Domain .$$

Therefore, given a term t of the form `Domain.routing` that fails to be candidate, Dynamite explores whether `Domain2.routing` or `Domain3.routing` are indeed candidates. The technique is applied in lines 13–17.

(2) The witness candidate could then be the intersection of `t` with another term. We explore this possibility in lines 18–20.

Over-approximation can be checked with the aid of the Alloy Analyzer by checking the assertion

```
assert overApproximation { some v : t | alpha(v) }
```

5.6.7 *Witness Precandidate Failure by Under-Approximation (lines 22–23).* As explained in Section 5.6.6, term `t` may fail to be a witness precandidate because in some environment e it denotes a set that contains an object that does not satisfy α. Yet there might be an already discarded witness `t'` that satisfies α in environment e. We then explore if there is a subset of the discarded witnesses that jointly with `t` form a coverage. If a coverage exists, it is returned in variable *result*. A coverage is determined with the aid of the Alloy Analyzer by checking assertion

```
assert underApproximation {
  all v:univ | (v in t1 => alpha(v)) || ... || (v in tn => alpha(v))}
```

5.6.8 *Lack of Witness Precandidates (lines 29–33).* If a new witness precandidate is not found in the selected environment, it may be due to, essentially, four reasons:

(1) the existentially quantified formula is not true within the prescribed scopes,
(2) the bound on the complexity of terms considered is smaller than required (new witness precandidates might be found if the complexity bound were increased),
(3) formula α' is relativized, and therefore part of the available complexity is spent on the relativization term, and not enough complexity is left to build a new precandidate,
(4) the included strategies fail to produce a witness.

In the first case the lack of precandidates may be due to the exhaustion of all the witness precandidates, and the algorithm should terminate without returning a precandidate. Similarly, in the second case the algorithm should be run again but with an increase in the complexity bound. The third case will occur when formula α' is relativized. Therefore, we will remove the relativization in order to enable the search for further witness precandidates. In the fourth case, new strategies should be added to the algorithm.

5.6.9 *Termination and Correctness.* Termination is guaranteed because in each loop iteration a new witness precandidate must be generated; due to the bound on term complexity only finitely many terms can be generated. Correctness, understood as producing a witness regardless of the analysis scopes, cannot be achieved due to the undecidability of classical first-order logic. Therefore, the algorithm may produce witness candidates that are not suitable for finishing the proof. The effectiveness of the algorithm is evaluated experimentally below.

5.6.10 *Experimental Evaluation.* In this section we will present 4 examples on which we used Dynamite in order to generate witness candidates automatically. We extended Dynamite with a new command solve-inst that, given a sequent whose consequent is a single existentially quantified formula, returns a witness candidate. Besides bounding the total complexity of the generated candidates, it is also possible to bound the number of times each Alloy operator is allowed to occur in generated candidates. In all the experiments each Alloy operator (with the exception of the sequential composition, that was not bounded) was allowed to occur 0 or 1 times. We used a computer with the following configuration: Intel(R) Core(TM) i5 quad core CPU running at 2.67GHz, 8GB of RAM. The operating system was Debian GNU/Linux 6.0, running Kernel 2.6.32-5-amd64.

5.6.10.1 *Example 1.* When verifying assertion `BindingPreservesDeterminism` the following assertion had to be proved:

```
assert prop1 {some ai: Identifier |
      ai in i.^newBinding && a in ai.^(d.dstBinding)}
```

Dynamite retrieves as witness the term[10]

```
{((i.^newBinding):>(*((d.dstBinding)).a))},
```

which indeed allowed us to complete the proof. It took Dynamite 172 seconds to retrieve the witness.

5.6.10.2 *Example 2.* In [Jackson 2006, Appendix A], an exercise involving properties of binary relations is proposed. As part of the Alloy model, the following assertion is presented:

```
assert ReformulateNonEmptinessOK {
     all r: univ->univ |
        some r iff (some x, y: univ | x->y in r)
}
```

Let us consider the assertion obtained by:

—skolemizing the universal quantifier,

—substituting `iff` by `implies`, and

—making the antecedent of the implication (`some r`) a new hypothesis.

The resulting model then contains:

[10]Given a set S and a relation R, the Alloy terms S<:R and R:>S restrict the domain/codomain or relation R to set S, respectively.

```
one sig D {r : univ -> univ}
fact {some D.r}
assert ReformulateNonEmptinessOK {
        some x, y: univ | x->y in D.r
}
```

Notice that there are two quantified variables. Therefore, we applied Dynamite in order to provide first a witness for the outer quantification. Dynamite returned term `(D.(r.univ))` in 161 seconds. Once the witness for the outer quantifier was found, we looked for a witness for the inner quantifier. Since term `(D.(r.univ))` denotes a set, Dynamite produces the following assertion in order to look for the inner witness:

```
fact {x1 in (D.(r.univ))}
assert ReformulateNonEmptinessOK {
        some y: univ | x1->y in D.r
}
```

Dynamite returns term `(x1.(x1<:(D.r)))` as the inner witness in 79 seconds. Using these witnesses the assertion is easily verified.

5.6.10.3 *Example 3.* In [Ramananandro 2008], as part of an Alloy model of the Mondex electronic purse, the following assertion was presented to be analyzed with the aid of the Alloy Analyzer:

```
assert Rbc_init {
   all c : ConWorld |
      ConInitState [c]
         implies some b : ConWorld {
            Rbc [b, c]
            BetwInitState [b]
         }
}
check Rbc_init for 10 but 2 ConState  -- 10007s
check Rbc_init for 10 but 10 ConState -- aborted by user after
                                      -- 7h computation [minisat]
```

According to the original model, it took the Alloy Analyzer 2.8 hours to analyze the assertion using 2 `ConState`, and the analysis was interrupted after 7 hours for a scope of 10 `ConState`. We verified this assertion using Dynamite in under 10 minutes. During the proof it was necessary to determine a witness for the existential quantifier in assertion `Rbc_init`. It took Dynamite 90 seconds to provide the correct witness.

5.6.10.4 *Example 4.* This example allows us to show a case in which Dynamite provides a non-atomic coverage as witness candidate. We present the Alloy model including assertion `coverSample` in Fig. 11. Running the witness candidate generator on assertion `coverSample` returned the coverage {`i,i2`}. Notice that term `i+i2` is not a solution due to fact `f2`. Let us consider the Alloy instance depicted in Fig. 12. Notice first that the instance is indeed a model for the specification. Also,

```
module fm06_extra2
open fm06_defs

sig Agent { }
abstract sig Identifier { }
sig Name, Address extends Identifier { }
sig AddressPair extends Identifier {
   addr: Address, name: Name
}
sig Domain {
   endpoints: set Agent, space: set Address, routing: space -> endpoints
}
sig Path {
   source: Address, dest: Address, generator: Agent, absorber: Agent
}
sig Domain2 extends Domain {
   dstBinding: Identifier -> Identifier
} { all i: Identifier |
      i in dstBinding.Identifier =>
         (  (i in Address => i in space) && (i in AddressPair => i.addr in space))
}
sig Path2 extends Path { origDst: Identifier }
one sig a in Address {}
one sig d in Domain2 {}
sig NB in Identifier { newBinding: set Identifier }
one sig i in Identifier {}
one sig i2 in Identifier {}
one sig i3 in Identifier {}

fact { all disj p1, p2: AddressPair | p1.addr != p2.addr || p1.name != p2.name }
fact { some Agent }
fact { some Identifier }
fact { some Domain }
fact { some Path }
fact f1 { some i.^newBinding }
fact f2 { #(i.^newBinding + i2.^newBinding) = 2 }
fact f3 { some i2.^newBinding }
fact f4 { i.^newBinding != i2.^newBinding }

assert coverSample { some x: Identifier | one x.^newBinding }
check coverSample for 6 but 2 Domain
```

Fig. 11. A sample model leading to a non-atomic coverage witness.

the instance shows that `i2` alone cannot be a witness candidate. If we permute `i2` and `i` in Fig. 12, we see that `i` cannot be a witness candidate. It took Dynamite 16 seconds to provide the witness.

5.6.11 *Limitations.* This technique heavily relies on the model-finding ability of the Alloy Analyzer. Consequently, it shares the same limitations. When the search for a candidate begins, limits on the search space must are established by fixing a scope. Unlike standard Alloy models where the scope only constrains the explored semantic environments, Alloy models used for witness generation internalize Alloy's syntax as well. Therefore, besides providing scopes for data domains, we must also

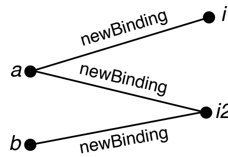


Fig. 12. An Alloy instance for assertion `coverSample`.

provide scopes for syntactic domains (maximum amount of occurrences of each operator symbol in the candidates, for example). If the limits imposed to the search are too restrictive, no admissible candidate will be generated. Otherwise, if the limits are too lax, the search can take too much time or lead to an *out of memory* exception.

6. CONCLUSIONS AND FURTHER WORK

The overall experience of proving theorems using Dynamite was very positive. It is remarkable that, although the crucial parts of the proofs are still relying on the user, using the Alloy Analyzer during the proving process proved to be useful in many ways:

—early detection of errors during key-steps of proofs helped us save time,

—the counterexamples retrieved by the Alloy Analyzer helped us improving our understanding of the problem domain,

—having leaner sequents helped us focusing on the right proof strategies,

—using the Alloy language during proofs contributed to smoothing the learning curve,

—automatically finding witnesses for existentially quantified assertions allowed us to shift the focus to higher-level proof strategies.

The work reported in this article revealed also some limitations of Dynamite in its current state. In the first place, the automation in the proving process is scarce. Only few proof steps are automatically solved (for instance, those referring to typing of relations or to witness candidate generation). Another limitation this work revealed was the need for an easily portable knowledge base that, as a library of predefined lemmas, allows the use of known general properties in the user-specific proofs. These are areas in which we are currently working.

REFERENCES

Andoni, A., Daniliuc, D., Khurshid, S. and Marinov, D. 2004. *Evaluating the "Small Scope Hypothesis"*, unpublished. Downloadable from `http://sdg.csail.mit.edu/publications.html`.

Arkoudas K. 2001. *Type-ω DPLs.* MIT AI Memo 2001-27.

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.

Beckert, B., Hahnle, R., and Schmitt, P.H. (eds.). *Verification of Object-Oriented Software: The KeY Approach*. Springer-Verlag (2007).

Bertot, Y. and Castéran, P. 2004. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*, EATCS Texts in Theoretical Computer Science.

Blanchette, J. and Nipkow T. 2010. *Nitpick: A counterexample generator for higher-order logic based on a relational model finder*. Proceedings of the First International Conference on Interactive Theorem Proving (ITP 2010), Lecture Notes in Computer Science 6172, 131–146, Springer.

Clarke, E., Grumberg, O. and Peled, D. 1999. *Model Checking*, MIT Press.

Mendonça de Moura L., and Bjorner N.. *Z3: An Efficient SMT Solver*. Proceedings of Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2008, Lecture Notes in Computer Science 4963, 337–340, Springer.

El Ghazi, A. and Taghdiri, M., 2011. *Relational Reasoning via SMT Solving*. Proceedings of Formal Methods (FM) 2011, Lecture Notes in Computer Science 6664, 133-148, Springer.

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., Haeberer A.M. and Veloso P.A.S. 1997. *A Finite Axiomatization for Fork Algebras*, Logic Journal of the IGPL, Vol. 5, No. 3, 311–319.

Frias M.F., López Pombo C.G. and Aguirre N. 2004. *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, pp. 162–175.

Frias, M.F., López Pombo, C.G. and Moscato, M.M. 2007. *Alloy Analyzer+PVS in the analysis and verification of Alloy specifications*. In Grumberg, O., Huth, M., eds.: Proceedings of the 13th. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007). Volume 4424 of Lecture Notes in Computer Science., Braga, Portugal, Springer-Verlag, pp. 587–601.

Gentzen, G. 1935. Untersuchungen über das logische Schließen. Mathematische Zeitschrift, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North–Holland, 1969.

Gordon, M. J. C. 1989. *Mechanizing Programming Logics in Higher Order Logic*. Current Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, 1989.

Holzmann, G. J. 2003. *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley.

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*. Association for the Computer Machinery, ACM Press, Vienna, Austria, 62–73.

Jackson, D. 2006. *Software Abstractions*. The MIT Press.

Maddux, R.D. 1991. *Pair-Dense Relation Algebras*, Transactions of the AMS, Vol. 328, N. 1.

Kaufmann, T., McConnell, C., Vazquez, I., Antoniotti, M., Campbell, R., and Amoroso, P. 2002. *ILISP User Manual*, Available at http://sourceforge.net/projects/ilisp/.

Moscato, M., López Pombo, C. G., and Frias, M. F. 2010. *Dynamite 2.0: New Features Based on UnSAT-Core Extraction to Improve Verification of Software Requirements*, International Conference on Theoretical Aspects of Computing (ICTAC) 2010. Lecture Notes in Computer Science 6255, Springer-Verlag, Berlin, Germany, 275–289.

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.

Owre, S. 2008. *A brief overview of the PVS user interface*, 8th International Workshop User Interfaces for Theorem Provers (UITP08), Montreal, Canada.

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

Owre, S., Rushby, J. M., and Shankar, N., 1992. *PVS: A Prototype Verification System*, Proceedings of the 11th International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence 607, 748–752, Springer.

Ramananandro T. 2008. *Mondex, an electronic purse: specification and refinement checks with the Alloy model-finding method*, Formal Aspects of Computing, 20(1), January 2008, pp. 21–39.

Torlak E., Chang F., and Jackson D. 2008. *Finding Minimal Unsatisfiable Cores of Declarative Specifications.* Proceedings of Formal Methods 2008, Lecture Notes in Computer Science 5014, 326–341, Springer.

Ulbrich M., Geilmann U., El Ghazi A., and Taghdiri M. 2012. *A Proof Assistant for Alloy Specifications.* Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2012, Lecture Notes in Computer Science 7214, 422–436, Springer.

Zave, P. 2006. *Compositional binding in network domains.* In Misra, J., Nipkow, T., Sekerinski, E., eds.: Proceedings of Formal Methods 2006: the 14th. International FME Symposium. Volume 4085 of Lecture Notes in Computer Science., Hamilton, Canada, Springer-Verlag (2006) 332–347

A. PROOFS OF LEMMAS AND THEOREMS

Given an Alloy environment, we can characterize those proper PDOCFA that are candidates to interpret the Alloy environment.

Definition A.1. Let e be an Alloy environment. A proper PDOCFA $\mathfrak{F}$ is compatible with environment e if the relational environment e_{P} (as defined in Def. 3.6), is correctly defined in $\mathfrak{F}$.

In Def. A.1, *correctly defined* means that for each symbol s, $e'(s)$ yields a relation in $\mathfrak{F}$.

From the previous definitions, the following lemma can be proved by induction on the structure of Alloy terms. The lemma, besides being necessary for the proof of interpretability, shows also in what sense the previous constructions can be considered "canonical". Notice that both Alloy and relational environments can be homomorphically extended to functions assigning appropriate values to complex terms built from the constants of each language. For the sake of simplifying the notation we will use the same notation for environments and their homomorphic extensions.

Lemma A.2. *Let e be an Alloy environment. Let $\mathfrak{F}_e$ be a PDOCFA compatible with environment e according to Def. A.1. Then, for every Alloy term t such that $e(t) \subseteq e(sig_{i_1}) \times \cdots \times e(sig_{i_k})$, we have:*

$$e_{\mathsf{P}}(T(t)) = \begin{cases} \{\, \langle a, a \rangle : a \in e(t) \,\}\,, & \text{if } k = 1 \\ \{\, \langle a_1, a_2 \star \cdots \star a_k \rangle : \langle a_1, a_2, \ldots, a_k \rangle \in e(t) \,\}\,. & \text{if } k > 1 \end{cases}$$

Proof. The proof follows by induction on the structure of the Alloy term t. The proof is trivial for the constants *iden*, *univ* and *none*. We present detailed proofs for the cases in which t is an individual variable or $t = t_1.t_2$. The other cases are easier.

—if t is the individual variable v:

$$\begin{aligned} e_{\mathsf{P}}(T(t)) &= e_{\mathsf{P}}(T(v)) && \text{(by Def. } t\text{)}\\ &= e_{\mathsf{P}}(V) && \text{(by Def. } T\text{)}\\ &= \{\, \langle e(v), e(v) \rangle \,\} && \text{(by Def. 3.6)}\\ &= \{\, \langle a, a \rangle : a \in e(t) \,\} \ . && \text{(by set theory and Def. } t\text{)} \end{aligned}$$

—if $t = t_1.t_2$: Since both the result of the lemma and the definition of $\bullet$ are given by cases, we will consider 6 different cases depending on k_1 (the rank of t_1) and k_2 (the rank of t_2). Following the typing constraints of Alloy, navigation is not defined when $k_1 = k_2 = 1$.

—$k_1 = 1$ and $k_2 = 2$ (then, $k = 1$):

$$\begin{aligned} e_{\mathsf{P}}(T(t)) &= e_{\mathsf{P}}(T(t_1.t_2)) && \text{(by Def. } t\text{)}\\ &= e_{\mathsf{P}}(T(t_1) \bullet T(t_2)) && \text{(by Def. } T\text{)}\\ &= e_{\mathsf{P}}(Ran\,(T(t_1).T(t_2))) && \text{(by Def. } \bullet\text{)}\\ &= Ran\,(e_{\mathsf{P}}(T(t_1)).e_{\mathsf{P}}(T(t_2))) && \text{(by } e_{\mathsf{P}} \text{ homomorphism)}\\ &= Ran\,(\{\, \langle a, a \rangle : a \in e(t_1) \,\} \,.\, \{\, \langle a, b \rangle : \langle a, b \rangle \in e(t_2) \,\}) && \text{(by Ind. Hyp.)}\\ &= Ran\,(\{\, \langle a, b \rangle : a \in e(t_1) \ \wedge \ \langle a, b \rangle \in e(t_2) \,\}) && \text{(by Def. ``.'')}\\ &= \{\, \langle b, b \rangle : \exists a\,(a \in e(t_1) \ \wedge \ \langle a, b \rangle \in e(t_2)) \,\} && \text{(by Def. } Ran\text{)}\\ &= \{\, \langle b, b \rangle : b \in e(t_1).e(t_2) \,\} && \text{(by Def. ``.'')}\\ &= \{\, \langle b, b \rangle : b \in e(t_1.t_2) \,\} && \text{(by } e \text{ homomorphism)}\\ &= \{\, \langle b, b \rangle : b \in e(t) \,\} \ . && \text{(by Def. } t\text{)} \end{aligned}$$

—$k_1 = 1$ and $k_2 > 2$:
Notice that

$$\begin{aligned} & e_{\mathsf{P}}(T(t_1)).e_{\mathsf{P}}(T(t_2))\\ &= \{\, \langle a, a \rangle : a \in e(t_1) \,\}\\ &\quad .\, \{\, \langle b_1, b_2 \star \cdots \star b_{k_2} \rangle : \langle b_1, b_2, \ldots, b_{k_2} \rangle \in e(t_2) \,\} && \text{(by Ind. Hyp.)}\\ &= \{\, \langle b_1, b_2 \star \cdots \star b_{k_2} \rangle : b_1 \in e(t_1) \wedge \langle b_1, b_2, \ldots, b_{k_2} \rangle \in e(t_2) \,\} \ . && \text{(by Def. ``.'')} \end{aligned}$$

Then,

$$\begin{aligned} & Ran\,(e_{\mathsf{P}}(T(t_1)).e_{\mathsf{P}}(T(t_2)))\\ &= \{\, \langle b_2 \star \cdots \star b_{k_2}, b_2 \star \cdots \star b_{k_2} \rangle : \exists b_1\,(b_1 \in e(t_1) \wedge \langle b_1, b_2, \ldots, b_{k_2} \rangle \in e(t_2)) \,\}\\ & && \text{(by Def. } Ran\text{)}\\ &= \{\, \langle b_2 \star \cdots \star b_{k_2}, b_2 \star \cdots \star b_{k_2} \rangle : \langle b_2, \ldots, b_{k_2} \rangle \in e(t_1).e(t_2) \,\} && \text{(by Def. ``.'')}\\ &= \{\, \langle b_2 \star \cdots \star b_{k_2}, b_2 \star \cdots \star b_{k_2} \rangle : \langle b_2, \ldots, b_{k_2} \rangle \in e(t_1.t_2) \,\} && \text{(by } e \text{ homo.)}\\ &= \{\, \langle b_2 \star \cdots \star b_{k_2}, b_2 \star \cdots \star b_{k_2} \rangle : \langle b_2, \ldots, b_{k_2} \rangle \in e(t) \,\} \ . && \text{(by Def. } t\text{)} \end{aligned}$$

From the definitions of π and ρ, we can reason

$$\begin{aligned} & \sim \pi \,.\, Ran\,(e_{\mathsf{P}}(T(t_1)).e_{\mathsf{P}}(T(t_2)))\ .\,\rho\\ &\qquad = \{\, \langle b_2, b_3 \star \cdots \star b_{k_2} \rangle : \langle b_2, b_3, \ldots, b_{k_2} \rangle \in e(t) \,\} \ . \qquad (4) \end{aligned}$$

Joining the previous proofs we obtain:

$$\begin{aligned}
e_{\mathsf{P}}(T(t)) &= e_{\mathsf{P}}(T(t_1.t_2)) && \text{(by Def. } t\text{)}\\
&= e_{\mathsf{P}}(T(t_1) \bullet T(t_2)) && \text{(by Def. } T\text{)}\\
&= e_{\mathsf{P}}(\sim \pi \,.\, Ran\,(T(t_1).T(t_2))\,.\rho) && \text{(by Def. } \bullet\text{)}\\
&=\sim \pi \,.\, Ran\,(e_{\mathsf{P}}(T(t_1)).e_{\mathsf{P}}(T(t_2)))\,.\rho && \text{(by } e_{\mathsf{P}} \text{ homomorphism)}\\
&= \{\, \langle b_2, b_3 \star \cdots \star b_{k_2}\rangle : \langle b_2, b_3, \ldots, b_{k_2}\rangle \in e(t)\,\} \ . && \text{(by (4))}
\end{aligned}$$

—$k_1 = 2$ and $k_2 = 1$ (then, $k = 1$):

$$\begin{aligned}
e_{\mathsf{P}}(T(t)) &= e_{\mathsf{P}}(T(t_1.t_2)) && \text{(by Def. } t\text{)}\\
&= e_{\mathsf{P}}(T(t_1) \bullet T(t_2)) && \text{(by Def. } T\text{)}\\
&= e_{\mathsf{P}}(Dom\,(T(t_1).T(t_2))) && \text{(by Def. } \bullet\text{)}\\
&= Dom\,(e_{\mathsf{P}}(T(t_1)).e_{\mathsf{P}}(T(t_2))) && \text{(by } e_{\mathsf{P}} \text{ homomorphism)}\\
&= Dom\,(\{\, \langle a,b\rangle : \langle a,b\rangle \in e(t_1)\,\} \,.\, \{\, \langle b,b\rangle : b \in e(t_2)\,\}) && \text{(by Ind. Hyp.)}\\
&= Dom\,(\{\, \langle a,b\rangle : \langle a,b\rangle \in e(t_1) \ \wedge\ b \in e(t_2)\,\}) && \text{(by Def.``.'')}\\
&= \{\, \langle a,a\rangle : \exists b\,(\langle a,b\rangle \in e(t_1) \ \wedge\ b \in e(t_2))\,\} && \text{(by Def. } Dom\text{)}\\
&= \{\, \langle a,a\rangle : a \in e(t_1).e(t_2)\,\} && \text{(by Def. ``.'')}\\
&= \{\, \langle a,a\rangle : a \in e(t_1.t_2)\,\} && \text{(by } e \text{ homomorphism)}\\
&= \{\, \langle a,a\rangle : a \in e(t)\,\} \ . && \text{(by Def. } t\text{)}
\end{aligned}$$

—$k_1 = 2$ and $k_2 > 1$:

$$\begin{aligned}
e_{\mathsf{P}}(T(t)) &= e_{\mathsf{P}}(T(t_1.t_2)) && \text{(by Def. } t\text{)}\\
&= e_{\mathsf{P}}(T(t_1) \bullet T(t_2)) && \text{(by Def. } T\text{)}\\
&= e_{\mathsf{P}}(T(t_1).T(t_2)) && \text{(by Def. } \bullet\text{)}\\
&= e_{\mathsf{P}}(T(t_1)).e_{\mathsf{P}}(T(t_2)) && \text{(by } e_{\mathsf{P}} \text{ homomorphism)}\\
&= \{\, \langle a,a_1\rangle : \langle a,a_1\rangle \in e(t_1)\,\}\\
&\qquad .\, \{\, \langle a_1, a_2 \star \cdots \star a_{k_2}\rangle : \langle a_1, a_2, \ldots, a_{k_2}\rangle \in e(t_2)\,\} && \text{(by Ind. Hyp.)}\\
&= \{\langle a, a_2 \star \cdots \star a_{k_2}\rangle : \exists a_1\,(\langle a,a_1\rangle \in e(t_1)\\
&\qquad \wedge\ \langle a_1, a_2, \ldots, a_{k_2}\rangle \in e(t_2))\} && \text{(by Def. ``.'')}\\
&= \{\, \langle a, a_2 \star \cdots \star a_{k_2}\rangle : \langle a, a_2, \ldots, a_{k_2}\rangle \in e(t_1).e(t_2)\,\} && \text{(by Def. ``.'')}\\
&= \{\, \langle a, a_2 \star \cdots \star a_{k_2}\rangle : \langle a, a_2, \ldots, a_{k_2}\rangle \in e(t_1.t_2)\,\} && \text{(by } e \text{ homo.)}\\
&= \{\, \langle a, a_2 \star \cdots \star a_{k_2}\rangle : \langle a, a_2, \ldots, a_{k_2}\rangle \in e(t)\,\} \ . && \text{(by Def. } t\text{)}
\end{aligned}$$

—$k_1 > 2$ and $k_2 = 1$:

$$\begin{aligned}
e_{\mathsf{P}}(T(t)) &= e_{\mathsf{P}}(T(t_1.t_2)) && \text{(by Def. } t\text{)}\\
&= e_{\mathsf{P}}(T(t_1) \bullet T(t_2)) && \text{(by Def. } T\text{)}\\
&= e_{\mathsf{P}}\,(T(t_1).\,(iden \otimes (\cdots \otimes ((iden \otimes T(t_2))\,.\pi)))) && \text{(by Def. } \bullet\text{)}\\
&= e_{\mathsf{P}}(T(t_1)).\,(iden \otimes (\cdots \otimes ((iden \otimes e_{\mathsf{P}}(T(t_2)))\,.\pi))) \ . && \text{(by } e_{\mathsf{P}} \text{ homo.)}
\end{aligned}$$

Notice that

$$iden \otimes e_{\mathsf{P}}(T(t_2)) = \{\, \langle a \star b, a \star b \rangle : \langle b, b \rangle \in e_{\mathsf{P}}(T(t_2)) \,\} \qquad \text{(by Def. } \otimes\text{)}$$
$$= \{\, \langle a \star b, a \star b \rangle : b \in e(t_2) \,\} \ . \qquad \text{(by Ind. Hyp.)}$$

Then,

$$(iden \otimes e_{\mathsf{P}}(T(t_2)))\,.\pi = \{\, \langle a \star b, a \rangle : b \in e(t_2) \,\} \ .$$

Therefore,

$$iden \otimes (\cdots \otimes (iden \otimes e_{\mathsf{P}}(T(t_2)))\,.\pi) =$$
$$\{\, \langle a_1 \star \cdots \star a_{k_1-3} \star a \star b, a_1 \star \cdots \star a_{k_1-2} \star a \rangle : b \in e(t_2) \,\} \ . \qquad (5)$$

By inductive hypothesis,

$$e_{\mathsf{P}}(T(t_1)) = \{\, \langle b_1, b_2 \star \cdots \star b_{k_1} \rangle : \langle b_1, b_2, \ldots, b_{k_1} \rangle \in e(t_1) \,\} \ . \qquad (6)$$

From (5) and (6), $\langle c_1, c_2 \star \cdots \star c_{k_1-1} \rangle \in e_{\mathsf{P}}(T(t))$ iff there exists (due to the definition of composition of binary relations) an object $d_1 \star \cdots \star d_{k_1-1}$ such that:
—$\langle c_1, d_1 \star \cdots \star d_{k_1-1} \rangle \in e_{\mathsf{P}}(T(t_1))$, (or, equivalently, $\langle c_1, d_1, \ldots, d_{k_1-1} \rangle \in e(t_1)$),
—$d_{k_1-1} \in e(t_2)$, and
—$d_i = c_{i+1}$ for $1 \leq i \leq k_1 - 2$.
From the previous conditions,

$$\langle c_1, c_2 \star \cdots \star c_{k_1-1} \rangle \in e_{\mathsf{P}}(T(t))$$
$$\textit{iff } \exists d_{k_1-1}\, (\langle c_1, c_2, \ldots, c_{k_1-1}, d_{k_1-1} \rangle \in e(t_1) \text{ and } d_{k_1-1} \in e(t_2))$$
$$\textit{iff } \langle c_1, c_2, \ldots, c_{k_1-1} \rangle \in e(t_1)\,.\,e(t_2)$$
$$\textit{iff } \langle c_1, c_2, \ldots, c_{k_1-1} \rangle \in e(t_1\,.\,t_2)$$
$$\textit{iff } \langle c_1, c_2, \ldots, c_{k_1-1} \rangle \in e(t) \ .$$

Thus, $e_{\mathsf{P}}(T(t)) = \{\, \langle c_1, c_2 \star \cdots \star c_{k_1-1} \rangle : \langle c_1, c_2, \ldots, c_{k_1-1} \rangle \in e(t) \,\}$.
—$k_1 > 2$ and $k_2 > 1$:

$$e_{\mathsf{P}}(T(t)) = e_{\mathsf{P}}(T(t_1\,.\,t_2)) \qquad \text{(by Def. } t\text{)}$$
$$= e_{\mathsf{P}}(T(t_1) \bullet T(t_2)) \qquad \text{(by Def. } T\text{)}$$
$$= e_{\mathsf{P}}(T(t_1)) \bullet e_{\mathsf{P}}(T(t_2)) \ . \qquad \text{(by } e_{\mathsf{P}} \text{ homo.)}$$
$$= e_{\mathsf{P}}\,(T(t_1).\,(iden \otimes (\cdots \otimes (iden \otimes T(t_2)))))) \qquad \text{(by Def. } \bullet\text{)}$$
$$= e_{\mathsf{P}}(T(t_1)).\,(iden \otimes (\cdots \otimes (iden \otimes e_{\mathsf{P}}(T(t_2))))) \qquad \text{(by } e_{\mathsf{P}} \text{ homo.)}$$

By inductive hypothesis,

$$e_{\mathsf{P}}(T(t_1)) = \{\, \langle a_1, a_2 \star \cdots \star a_{k_1} \rangle : \langle a_1, a_2, \ldots, a_{k_1} \rangle \in e(t_1) \,\} \qquad (7)$$

and

$$e_{\mathsf{P}}(T(t_2)) = \{\, \langle b_1, b_2 \star \cdots \star b_{k_2} \rangle : \langle b_1, b_2, \ldots, b_{k_2} \rangle \in e(t_2) \,\} \ . \qquad (8)$$

From (8) and Def. $\otimes$,

$$\begin{aligned} &iden \otimes (\cdots \otimes (iden \otimes e_{\mathsf{P}}(T(t_2)))) \\ &\quad = \{\langle a_1 \star \cdots \star a_{k_1-2} \star b_1, a_1 \star \cdots \star a_{k_1-2} \star b_2 \star \cdots \star b_{k_2} \rangle : \\ &\qquad\qquad \langle b_1, b_2, \ldots, b_{k_2} \rangle \in e(t_2)\} \ . \end{aligned} \tag{9}$$

From (7) and (9), $\langle c_1, c_2 \star \cdots \star c_{k_1+k_2-2} \rangle \in e_{\mathsf{P}}(T(t))$ iff there exists (due to the definition of composition of binary relations) $d_1, \ldots, d_{k_1-1}$ such that:
—$\langle c_1, d_1 \star \cdots \star d_{k_1-1} \rangle \in e_{\mathsf{P}}(T(t_1))$ (or, equivalently, $\langle c_1, d_1, \ldots, d_{k_1-1} \rangle \in e(t_1)$),
—$\langle d_{k_1-1}, c_{k_1}, \ldots, c_{k_1+k_2-2} \rangle \in e(t_2)$, and
—$d_i = c_{i+1}$ for $1 \leq i \leq k_1 - 2$.
From the previous conditions,

$$\begin{aligned} & \langle c_1, c_2 \star \cdots \star c_{k_1+k_2-2} \rangle \in e'(T(t)) \\ \textit{iff}\ & \exists d_{k_1-1} \left(\langle c_1, c_2, \ldots, c_{k_1-1}, d_{k_1-1} \rangle \in e(t_1) \right. \\ & \quad \left. \text{and } \langle d_{k_1-1}, c_{k_1}, \ldots, c_{k_1+k_2-2} \rangle \in e(t_2) \right) \\ \textit{iff}\ & \langle c_1, c_2, \ldots, c_{k_1+k_2-2} \rangle \in e(t_1) . e(t_2) \\ \textit{iff}\ & \langle c_1, c_2, \ldots, c_{k_1+k_2-2} \rangle \in e(t_1 . t_2) \\ \textit{iff}\ & \langle c_1, c_2, \ldots, c_{k_1+k_2-2} \rangle \in e(t) \ . \end{aligned}$$

Thus, $e_{\mathsf{P}}(T(t)) = \{ \langle c_1, c_2 \star \cdots \star c_{k_1+k_2-2} \rangle : \langle c_1, c_2, \ldots, c_{k_1+k_2-2} \rangle \in e(t) \}$.

■

LEMMA A.3. *Given a proper* PDOCFA $\mathfrak{F}$ *and a relational environment* e, *the Alloy environment* e_{A} *built according to Def. 3.7 satisfies for every Alloy term* t *with* $e_{\mathsf{A}}(t) \subseteq e_{\mathsf{A}}(sig_{i_1}) \times \cdots \times e_{\mathsf{A}}(sig_{i_k})$:

—*if* $k = 1$, $e_{\mathsf{A}}(t) = \{ a : \langle a, a \rangle \in e(T(t)) \}$,
—*if* $k > 1$, $e_{\mathsf{A}}(t) = \{ \langle a_1, a_2, \ldots, a_k \rangle : \langle a_1, a_2 \star \cdots \star a_k \rangle \in e(T(t)) \}$.

PROOF. By Def. 3.7, e_{A} satisfies:

—$e_{\mathsf{A}}(\text{sig}_i) = \left\{ a : \langle a, a \rangle \in e(iden_{\text{sig}_i}) \right\}$,
—$e_{\mathsf{A}}(R) = \{ \langle a_1, \ldots, a_n \rangle : \langle a_1, a_2 \star \cdots \star a_n \rangle \in e(\mathsf{R}) \}$,
—$e_{\mathsf{A}}(v_i) = a$ such that $e(v_i) = \{ \langle a, a \rangle \}$.

From the definition of e_{A}, e satisfies:

—$e(iden_{\text{sig}_i}) = \{ \langle a, a \rangle : a \in e_{\mathsf{A}}(\text{sig}_i) \}$,
—$e(\mathsf{R}) = \{ \langle a_1, a_2 \star \cdots \star a_n \rangle : \langle a_1, \ldots, a_n \rangle \in e_{\mathsf{A}}(R) \}$,
—$e(v_i) = \{ \langle a, a \rangle \}$ such that $e_{\mathsf{A}}(v_i) = a$.

We now have an Alloy environment e_{A} and a relational environment e that satisfy the conditions in Def. 3.6. Notice then that $\mathfrak{F}$ is compatible with the Alloy environment e_{A}. From Lemma A.2,

—if $k = 1$, $e(T(t)) = \{ \langle a, a \rangle : a \in e_{\mathsf{A}}(t) \}$,
—if $k > 1$, $e(T(t)) = \{ \langle a_1, a_2 \star \cdots \star a_k \rangle : \langle a_1, a_2, \ldots, a_k \rangle \in e_{\mathsf{A}}(t) \}$.

It then follows that

—if $k = 1$, $e_{\mathsf{A}}(t) = \{ a : \langle a, a\rangle \in e(T(t)) \}$,
—if $k > 1$, $e_{\mathsf{A}}(t) = \{ \langle a_1, a_2, \ldots, a_k\rangle : \langle a_1, a_2 \star \cdots \star a_k\rangle \in e(T(t)) \}$.

■

The following lemma will be used in the proof of Lemma A.5.

LEMMA A.4. *Let e be an Alloy environment. Let e_{P} be a relational environment defined according to Def. 3.6. Let $\mathfrak{F}_e$ be a* PDOCFA *compatible with environment e (c.f. Def. A.1). Let $r = \{ \langle a, a\rangle \} \subseteq iden_S$ be a point. Let $\mathfrak{F}_{e[x\mapsto a]}$ be a* PDOCFA *compatible with environment $e[x \mapsto a]$. Then,*

$$\mathfrak{F}_e \models \beta\,[e_{\mathsf{P}}[x \mapsto r]] \quad \textit{iff} \quad \mathfrak{F}_{e[x\mapsto a]} \models \beta\,[(e[x \mapsto a])_{\mathsf{P}}]$$

PROOF. In order to prove the lemma it suffices to show that $\mathfrak{F}_e = \mathfrak{F}_{e[x\mapsto a]}$ and $e_{\mathsf{P}}[x \mapsto r] = (e[x \mapsto a])_{\mathsf{P}}$. According to Defs. 3.6 and A.1, the construction of algebra $\mathfrak{F}_e$ does not depend on the value e assigns to variables. Therefore, it is immediate that $\mathfrak{F}_e = \mathfrak{F}_{e[x\mapsto a]}$. For all signatures, fields and variables distinct of x, it is clear that $e_{\mathsf{P}}[x \mapsto r]$ and $(e[x \mapsto a])_{\mathsf{P}}$ agree. For variable x we have:

$$e_{\mathsf{P}}[x \mapsto r](x) = r = \{ \langle a, a\rangle \} \ .$$

Similarly, by Def. 3.6,

$$(e[x \mapsto a])_{\mathsf{P}}(x) = \{ \langle e[x \mapsto a](x), e[x \mapsto a](x)\rangle \} = \{ \langle a, a\rangle \} \ .$$

■

LEMMA A.5. *Let e be an Alloy environment. Let e_{P} be defined according to Def. 3.6. Let $\mathfrak{F}$ be a* PDOCFA *compatible with environment e (c.f. Def. A.1). Then,*

$$\models \varphi[e] \iff \mathfrak{F} \models F(\varphi)[e_{\mathsf{P}}] \ .$$

PROOF. The proof proceeds by induction on the structure of the Alloy formula φ. We will concentrate on formulas built from atomic formulas (inclusions) and existential quantification. The other cases are easier.

—$\varphi = t_1$ in t_2:
For φ to be well-formed, t_1 and t_2 must stand for relations of the same arity.

$$\begin{aligned} & \mathfrak{F} \models F(t_1 \text{ in } t_2)[e_{\mathsf{P}}] & \\ \textit{iff}\ & \mathfrak{F} \models T(t_1) \text{ in } T(t_2)[e_{\mathsf{P}}] & \text{(by Def. 3.8)} \\ \textit{iff}\ & e_{\mathsf{P}}(T(t_1)) \subseteq e_{\mathsf{P}}(T(t_2)) \ . & \text{(by Def. of } \models) \end{aligned}$$

There are now two possibilities, namely, either both t_1, t_2 have arity 1, or they both have arity $k > 2$. Let us continue the proof for the unary case, being the remaining case similar. We then have

$$\begin{aligned} & e_{\mathsf{P}}(T(t_1)) \subseteq e_{\mathsf{P}}(T(t_2)) & \\ \textit{iff}\ & \{ \langle a, a\rangle : a \in e(t_1) \} \subseteq \{ \langle a, a\rangle : a \in e(t_2) \} & \text{(by Lemma A.2)} \\ \textit{iff}\ & e(t_1) \subseteq e(t_2) & \text{(by set-theory)} \\ \textit{iff}\ & \models t_1 \text{ in } t_2[e] \ . & \text{(by Def. of } \models) \end{aligned}$$

—$\varphi = \text{some } x : S \mid \alpha$:

$$
\begin{aligned}
& \mathfrak{F} \models F(\text{some } x : S \mid \alpha)[e_{\mathsf{P}}] & \\
\textit{iff } & \mathfrak{F} \models \text{some } x : S \mid F(\alpha)[e_{\mathsf{P}}] & \text{(by Def. 3.8)} \\
\textit{iff } & \mathfrak{F} \models \text{some } x \mid \textit{Point}(x) \ \&\& \ x \text{ in } \textit{iden}_S \ \&\& \ F(\alpha)[e_{\mathsf{P}}] & \text{(by abbreviation)} \\
\textit{iff } & \text{there exists } r \text{ such that:} & \\
& \quad \mathfrak{F} \models \textit{Point}(x)\,[e_{\mathsf{P}}[x \mapsto r]] \text{ and} & \\
& \quad \mathfrak{F} \models x \text{ in } \textit{iden}_S\,[e_{\mathsf{P}}[x \mapsto r]] \text{ and} & \\
& \quad \mathfrak{F} \models F(\alpha)\,[e_{\mathsf{P}}[x \mapsto r]] \ . & \text{(by Def. } \models)
\end{aligned}
$$

Since r is a point in $\textit{iden}_S$, let us assume $r = \{ \langle a, a \rangle \}$ (for $a \in S$). Notice that algebra $\mathfrak{F}$ does not depend on the value e assigns to variables. Thus, $\mathfrak{F}$ is also compatible with environment $e[x \mapsto a]$. Moreover, from the proof of Lemma A.4, $(e[x \mapsto a])_{\mathsf{P}} = e_{\mathsf{P}}[x \mapsto r]$.
Thus,

$$
\begin{aligned}
& \text{there exists } r \text{ such that:} & \\
& \quad \mathfrak{F} \models \textit{Point}(x)\,[e_{\mathsf{P}}[x \mapsto r]] \text{ and} & \\
& \quad \mathfrak{F} \models x \text{ in } \textit{iden}_S\,[e_{\mathsf{P}}[x \mapsto r]] \text{ and} & \\
& \quad \mathfrak{F} \models F(\alpha)\,[e_{\mathsf{P}}[x \mapsto r]] & \text{(by Def. } \models) \\
\textit{iff } & \text{there exists } a \in S \text{ such that} & \\
& \quad \mathfrak{F} \models F(\alpha)\,[(e[x \mapsto a])_{\mathsf{P}}] & \text{(by previous comment)} \\
\textit{iff } & \text{there exists } a \in S \text{ such that} & \\
& \quad \models \alpha\,[e[x \mapsto a]] & \text{(by inductive hypothesis)} \\
\textit{iff } & \models \text{some } x : S \mid \alpha\,[e] \ . & \text{(by Def. } \models)
\end{aligned}
$$

■

LEMMA A.6. *Let $\mathfrak{F}$ be a proper* PDOCFA. *Let e be a relational environment. Then, there exists an Alloy environment e_{A} built according to Def. 3.7 such that for every Alloy formula φ,*

$$\mathfrak{F} \models F(\varphi)[e] \iff \models \varphi[e_{\mathsf{A}}] \ .$$

PROOF. The proof proceeds by induction on the structure of formula φ.

—$\varphi = t_1 \text{ in } t_2$:

$$
\begin{aligned}
\mathfrak{F} \models F(t_1 \text{ in } t_2)[e] & \iff \mathfrak{F} \models T(t_1) \text{ in } T(t_2)[e] & \text{(by Def. 3.8)} \\
& \iff e(T(t_1)) \subseteq e(T(t_2)) & \text{(by Def. } \models) \\
& \iff e_{\mathsf{A}}(t_1) \subseteq e_{\mathsf{A}}(t_2) & \text{(by Lemma A.3)} \\
& \iff \models t_1 \text{ in } t_2[e_{\mathsf{A}}] \ . & \text{(by Def. } \models)
\end{aligned}
$$

The remaining parts of the proof follow a structure similar to that of the proof of Lemma A.5. ■

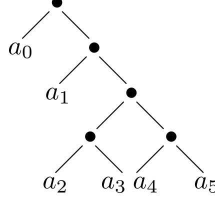
LEMMA A.7. *Given an Alloy environment e, there exists a proper* PDOCFA *$\mathfrak{F}$ compatible with e.*

PROOF. Assume the Alloy model declares signatures $Sig_1, \ldots, Sig_I$. Let $A = \bigcup_{1 \leq j \leq I} e(Sig_j)$. Let $Tree(A)$ be the smallest set satisfying the following conditions:

—$A \subseteq Tree(A)$, and

—$Tree(A) \times Tree(A) \subseteq Tree(A)$.

$Tree(A)$ describes finite binary trees with data from A in the leaves. For instance, element $\langle a_0, \langle a_1, \langle\langle a_2, a_3\rangle, \langle a_4, a_5\rangle\rangle\rangle\rangle \in Tree(A)$ describes the tree



Let us consider the PDOCFA $\mathfrak{F}$ with universe[11] $Pw(Tree(A) \times Tree(A))$. All the operators but fork have their standard set-theoretical meaning. For fork we define

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

In order to prove compatibility we must show that signatures and fields defined in the Alloy model are given appropriate values according to environment e_{P}. For signature Sig_j $(1 \leq j \leq I)$, $e_{\mathsf{P}}(Sig_j) = iden_{Sig_j}$, which clearly belongs to $Pw(Tree(A) \times Tree(A))$. For a field F declared as

```
sig S { F : S1->...->Sk }
```

the relation $e_{\mathsf{P}}(F)$ defined as

$$\{\langle s_0, s_1 \star \cdots \star s_k\rangle : \\ s_0 \in S \;\&\&\; s_1 \in S_1 \;\&\&\; \cdots \&\& \; s_k \in S_k \;\&\&\; \langle s_0, s_1, \ldots, s_k\rangle \in e(F)\}$$

belongs to $Pw(Tree(A) \times Tree(A))$ provided $a \star b$ is defined as $\langle a, b\rangle$.

■

[11]We denote by $Pw(X)$ the power set of set X.