Lessons Learned on the Verification of Models Using Dynamite

Mariano M. Moscato[1], Carlos G. Lopez Pombo[1,2], and Marcelo F. Frias[1,2]

[1] Department of Computer Science, School of Science, University of Buenos Aires
[2] CONICET

Abstract. The Dynamite Proving System (DPS) provides an assisted theorem prover for Alloy. In this paper we report our experiences on using DPS in the verification of an industrial size model for compositional bindings in network domains, developed by Pamela Zave at AT&T. We also review the DPS foundations, architecture, and some of its main features.

1 Introduction

Specification of software systems is considered a worthwhile activity in most modern software development processes. The main objective of specifying (or modeling) software systems is the possibility of describing software artifacts with a certain degree of abstraction. In this way analysis tasks can be performed on these formal descriptions. This analysis may allow one to discover properties of the specified artifacts, and understand the implications of our design decisions.

Alloy [1] is a formal modeling language that supports notations ubiquitous in object oriented programming, and whose semantics is based on relations. The existence of the Alloy Analyzer allows one to analyze Alloy specifications in a fully automatic way. To perform this analysis the Alloy Analyzer translates Alloy specifications (where domains are bounded to finite sizes) to a propositional formula. Then, this formula is tested for satisfiability using off-the-shelf SAT-solvers. Bounding the size of domains has a direct impact on the conclusions we can draw from the analysis process. If a counterexample for a given assertion is found, then the model is certainly flawed. On the other hand, if no counterexample is found, we can only conclude that no counterexamples exist within these domain sizes. Thus choosing larger bounds may show the existence of previously unforeseen errors. Notice that this limited analyzability offers the possibility of getting rid of most of the errors introduced in the modeling process. At the same time, models for critical applications cannot entirely rely only on this kind of analysis because these applications require a proof of the absence of certain errors.

An alternative is the use of heavyweight formal methods, as for instance semi-automatic theorem provers, and among these, PVS [2]. Theorem provers have limitations too. First, they require an expertise from the user that many times discourages their use. And second, they have their own languages which are seldom close to lightweight modeling languages. This may be a source of error introduction in case lightweight models have to be translated. In theorem proving, minor errors in a model may require to redo proofs that were using inappropriate hypotheses. Much the same as errors overlooked during software requirement elicitation have a greater impact the more advanced the development stage is, model errors have greater impact the more auxiliary lemmas have been introduced. In this context, a marriage between simple automatically analyzable formal modeling languages and semi-automatic theorem provers is in fact necessary when analyzing critical models.

In [3] we presented Dynamite in order to bridge the gap existing between these two techniques for the language Alloy. Dynamite is an extension of PVS that incorporates features such as:

– sound automatic translation of Alloy models to PVS (thus preventing the introduction of errors in the translation process),
– a complete proof calculus for Alloy (therefore, valid Alloy properties can be proved),
– modified PVS pretty-printer that shows proof steps in Alloy language (thus bridging the gap between Alloy and PVS), and
– fluid automatic interaction with the Alloy Analyzer in order to automatically analyze hypotheses introduced during the theorem proving process.

The goal of this paper is to report lesson learnt on using the tool for the verification of an model for compositional bindings in network domains developed by Pamela Zave for AT&T [4] used to state and validate meaningful properties of real life computer networks.

The paper is organized as follows, in Sec. 2 we describe DPS, in Sec. 3 we introduce the case study. Section 4 reports our experiences on proving properties of the case study explained in Sec. 3. In Sec. 5 we present empirical results, draw some conclusions and discuss related work. Finally, in Sec. 6, we list some further research directions.

2 The Dynamite Proving System

DPS is a tool that allows the user to write specifications (written in Alloy language), to validate these specifications (by finding models or counterexamples of user-defined assertions in bounded domains with the help of the Alloy Analyzer), and to prove assertions (using the theorem proving facilities provided by PVS).

We will now describe how these three activities are carried out in order to verify the model subject of this article . We will also include some notes on the theoretical foundations of DPS.

Writing an Alloy Model – The Alloy Language. DPS' frontend uses GNU Emacs [5], a highly customizable text editor, as its user interface. When the tool starts, a GNU Emacs window is opened and, within it, the user can write an Alloy model. We will now review some of the Alloy features.

In [4], Zave presents a formal model of compositional binding in network domains. In that context, *agents* (which might be hardware devices or other software systems) communicate with each other over a network *Domain*. These agents are called *endpoints*. Each domain has a set of *addresses*, called *space*, that must be used by the agents in their communications.

Different sorts of objects can be distinguished in the previous description. Sorts are declared in Alloy using signatures (akin to classes in object orientation).

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

Signature `Agent` denotes a unary relation (i.e., a set) of atomic objects. Signature `Domain` declares a set of objects *Domain* having three fields of different type. The field `endpoints` denotes a binary relation $endpoints \subseteq Domain \times Agent$. Notice that without the modifier `set` in the declaration of field `endpoints`, relation *endpoints* would be a total function instead of a relation. The infrastructure of the connection domain is represented by the ternary relation $routing \subseteq Domain \times Address \times Agent$, in which for each tuple $\langle d, i, g \rangle$ the address i is in the domain *space* of d and g is an *endpoint* of the domain d.

To send a message an endpoint must find the way to bind a known identifier with the identifier used by the domain to locate the receiver. There are three ways of making such a binding. The simplest scenario is in which the initiator (of the communication) does its own search of the binding, and then sends a message whose destination is the resulting identifier. The original identifier has no restrictions on it. Unrestricted identifiers will be referred as *names*. Now, names and addresses can be abstracted as subtypes of a single type *identifier*.

```
abstract sig Identifier { }
sig Name, Address extends Identifier { }
```

Signature extension allows us to model single inheritance between signatures. The `abstract` modifier indicates that `Identifier` is just a way to denote the union of the sets of atoms of sort *Name* and *Address*.

In another scenario the initiator sends the message with an address (as destination) which is mapped by *routing* to an agent that is not the intended receiver of the message, but that will forward it on its way to its destination. Such an agent is called a *handler*. The handler handles the message by looking up the binding of the address with the destination, thus changing the destination of the message, and forwarding it to its new destination.

In the third scenario, the original identifier has two parts: an address part and a name part. The first part is routed to a handler, just as in the previous scenario. The handler has access to the binding of the hole pair, and handles the message by changing the destination to the resulting identifier and forwarding it. These address/name pairs also can be abstracted as a kind of identifier.

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

In Alloy, terms are constructed in the same way that in first-order logic. Constants such as `iden` (denoting the binary identity), `univ` (denoting the set holding all the atomic elements) and `none` (the empty relation), are terms. Also user defined constants are terms. Variables are terms, and applications of operators such as `+` (denoting the union of relations), `&` (denoting intersection), `-` (denoting the difference of relations), `~` (denoting the transposition, which flips pairs $\langle x, y\rangle$ of a binary relation to $\langle y, x\rangle$), `*` (for reflexive-transitive closure), "`^`" (for transitive closure) and "`.`" for composition of relations, called *navigation* in Alloy, yield terms.

Then, formulas are built from a set of terms by applying predicates to them in order to obtain atomic formulas. The available predicates can be divided in two groups: the standard *built in* predicates (such as: `no` to indicate that a relation is empty, equality of two terms, `in` to denote the inclusion of a term in another, etc.) and the *user defined* specification specific predicates. Compound formulas are built by using the standard propositional connectors (`&&` for conjunction, `||` for disjunction, `=>` for implication, `not` for negation, etc.) and quantifications over the atomic sorts defined by the signatures.

Axioms are included in a model under the keyword `fact`. An axiom saying that "there are no distinct address pairs with the same name and address", is written as:

```
fact { all disj p1, p2: AddressPair |
   p1.addr != p2.addr || p1.name != p2.name }
```

The `disj` modifier singles out that `p1` and `p2` are distinct *AddressPair* atoms.

Validating an Alloy model – The Alloy Analyzer. Once a model has been provided, it can be analyzed by looking for counterexamples of properties that are expected to hold in the model. These properties are called *assertions*. For instance, the (not necessarily valid) assertion stating then "no endpoint has more than one address in a domain" can be written in Alloy as:

```
assert UniqueAddress { all d: Domain, a1, a2: d.space |
  a1 = a2 || no( a1.(d.routing) & a2.(d.routing) ) }
```

DPS allows the user to search for counterexamples in which domains are bounded up-to a certain number of elements, using the Alloy Analyzer. The Alloy Analyzer translates the model and the negation of the assertion to a propositional formula. Of course, the translation heavily depends on the bounds declared by the user. Once a propositional formula has been obtained, the Alloy Analyzer employs off-the-shelf SAT-solvers to test the formula for satisfiability. If the

SAT-solver judges the formula as satisfiable, then it means that there exists a valuation satisfying the axioms and the negation of the property thus providing a counterexample serving as a witness of the existence of a flaw in the model. This valuation is then mapped backwards to the first-order model to provide some insight of where to find the error. The study of the counterexample allows the user to correct the error and continue the search for errors in the model. This process can be seen as the *debugging* of the specification.

Proving Assertions – PVS. As we already mentioned, even when the analysis of the specification does not end up with a counterexample, it does not mean that the assertions are valid. It could be the case that finding a counterexample requires larger scopes. Critical applications require higher levels of confidence about the absence of errors, thus theorem proving is an alternative to fulfill this need. DPS allows the user to prove that an assertion follows from a specification.

DPS heavily relies on the "Prototype Verification System" (PVS)[2]. As it is mentioned in [6], PVS is a prototype environment for specification and verification, and it provides effective theorem proving support. PVS' language and Alloy's language differ, so when the user decides to prove certain assertion, DPS automatically writes PVS theories describing the fork algebraic translation of the given Alloy specification (to be explained in subsection 2.1), and then starts the PVS theorem prover on the theorem corresponding to that assertion.

PVS is based on a sequent calculus, thus proofs can be seen as a tree in which every node expresses a *proof goal* (i.e. a sequent). Each proof goal is a *sequent* consisting of a sequence of formulas called *antecedents* and a sequence of formulas called *consequents*. Naturally, the intuitive interpretation of a sequent is that the conjunction of the antecedents implies the disjunction of the consequents. When a sequent is recognized as true, then that branch of the tree is closed, and when all the branches are closed, the proof is considered complete.

Proof rules are implemented as *proof commands* that can be applied on any open branch. The application results in one or more nodes that are direct descendants of the one to which the proof command was applied. Some examples of PVS proof commands are:

- `case`: given a formula, splits the current branch into two branches. In one of the branches the formula is introduced as a new antecedent, and is placed as a proof obligation in the other branch of the proof;
- `lemma`: given a lemma name, introduces an instance of that lemma as a new antecedent in the sequent;
- `inst`: given a list of terms, reduces universally quantified formulas in the antecedent or existentially quantified formulas in the consequent by instantiating the quantified variables with those terms in the specified order.

DPS also extends the library of PVS proof commands by adding new features, such as: allowing to find counterexamples of the formula introduced with a `case` in order to get confidence on the validity of that formula; analyzing the actual proof goal (by invoking the Alloy Analyzer); automatically selecting possible superfluous sequents, that could be left behind in the proving process; etc. We will return to these features in Section 3.

During all the proving process DPS shows the sequents and accepts the arguments of the proof commands in Alloy language, and not in its fork algebraic translation (see subsection 2.1), in order to make the proving process appealing to an Alloy user. For example, if the pretty-printer were deactivated, the initial proof goal for the assertion `UniqueAddress`, presented before, would look like this:

```
fm06?UniqueAddress :

  |-------
{1}   (FORALL(fm06?Domain3?srcBinding: (fm06?Domain3?srcBinding),
           fm06?Path?source: (fm06?Path?source),
           fm06?Path?generator: (fm06?Path?generator),
           fm06?Path3?finSrc: (fm06?Path3?finSrc),
           fm06?Domain3?BdstBinding: (fm06?Domain3?BdstBinding),
           fm06?Path2?origDst: (fm06?Path2?origDst),
           fm06?Domain?space: (fm06?Domain?space),
           fm06?AddressPair?addr: (fm06?AddressPair?addr),
           fm06?AddressPair?name: (fm06?AddressPair?name),
           fm06?Domain?routing: (fm06?Domain?routing),
           fm06?Domain?endpoints: (fm06?Domain?endpoints),
           fm06?Domain2?dstBinding: (fm06?Domain2?dstBinding),
           fm06?Path?dest: (fm06?Path?dest),
           fm06?Path?absorber: (fm06?Path?absorber),
           fm06?Domain3?AdstBinding: (fm06?Domain3?AdstBinding)):
        (FORALL (d: (fm06?Domain),
         a1: Carrier |
            (Leq(a1, Navigation(d, fm06?Domain?space)) AND Atom(a1)),
         a2: Carrier |
            (Leq(a2, Navigation(d, fm06?Domain?space)) AND Atom(a2))):
           ((a1 = a2) OR
              None( product(
                Navigation(a1, Navigation(d, fm06?Domain?routing)),
                Navigation(a2, Navigation (d, fm06?Domain?routing)))))))
```

The pretty printer displays the same proof goal as follows:

```
fm06?UniqueAddress :

  |-------
{1}   (all d : Domain, a1 : (d . space), a2 : (d . space) |
           (a1 = a2) || no ((a1 . (d . routing)) & (a2 . (d . routing))))
```

2.1 Dynamite Proving System foundations

Dynamite was presented in [3] and one of its most relevant attributes is the extension of PVS with a complete calculus for Alloy. This calculus is obtained by translating Alloy specifications (theories) to fork algebraic theories. A semantics preserving translation ensures that an assertion α (semantically) follows from an Alloy specification Σ if (and only if) its translation $T(\alpha)$ can be proved from the translation of the theory using the complete calculus for fork algebras. In symbols, $\Sigma \models \alpha \iff \{\, T(\sigma) \mid \sigma \in \Sigma \,\} \vdash T(\alpha)$.

Formal semantics of Alloy assigns semantics to expressions and formulas in a given *environment*. An environment is a function that assigns sets to signatures, adequate relations to relational constants (those arising from signature fields), and values to variables over individuals. In this sense, given an Alloy specification Σ, a model for Σ is an environment e such that $e \models \Sigma$ (e satisfies all the formulas in Σ). In order to interpret Alloy models, we use the class *point-dense proper ω-closure fork algebras*. These algebras are structures whose domain is a set of binary relations built on top of a base set B. If we call R the domain of such an algebra (notice that $R \subseteq Pw\,(B \times B)$), R has to be closed under the following operations for sets: *union* ($\cup$), *intersection* ($\cap$), *complement* (denoted, for a binary relation r, by $\overline{r}$), the *empty* binary relation ($\emptyset$), and the *universal* binary relation $B \times B$, (usually denoted by 1).

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

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

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

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

We can characterize points as nonempty relations that satisfy the property $x;1;x \subseteq Id$. If we denote the inclusion relation by "in" (as in Alloy), the predicate "Point" defined by "Point(p) $\iff$ $p \neq \emptyset \wedge p;1;p \subseteq Id$" determines those relations that are points.

In [7] we presented the calculus ω-CCFAU, which is an extension of the calculus for fork algebras [8] with axioms and an infinitary proof rule characterizing the reflexive-transitive closure. Later, in [3] this calculus got its final shape getting closer to Alloy's language and was proved to be complete for the class of algebras PDOCFA. Also in [3] we presented a semantic preserving translation F of Alloy terms and formulas to first-order formulas over fork algebraic terms and proved that ω-CCFAU is complete for Alloy specifications. To do this we provided a way to build point-dense ω-closure fork algebras from Alloy models and vice-versa. These translations were used to prove the following lemmata and theorem

Lemma 1. *[3, Lemma 1]*
Given an Alloy environment e, there exists a PDOCFA $\mathfrak{F}_e$ *and a relational environment e' such that for every Alloy formula φ, $\models \varphi[e] \iff \mathfrak{F}_e \models F(\varphi)[e']$.*

Lemma 2. *[3, Lemma 1]*
Given a PDOCFA $\mathfrak{F}$ *and a relational environment e, there exists an Alloy environment e' such that for every Alloy formula φ, $\mathfrak{F} \models F(\varphi)[e] \iff \models \varphi[e']$.*

Theorem 1. *[3, Thm. 2]*
Let $\Sigma \cup \{\varphi\}$ be a set of Alloy formulas. Then,

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

3 Compositional Binding in Network Domains

In [3], in order to support the need for a tool like DPS, we developed a case-study based on the model presented by Zave in [9]. In this article we want to report our experience using DPS on the industrial size model [4] but also some of the lessons we learned during the process of verification.

We already started introducing the model in Section 2 when we presented Alloy's basic constructs. Now we will finish the presentation of the part of the model we were working on, and show how we used DPS to verify its correctness.

The binding of any identifier results in another identifier that could require further binding. In that sense, binding is an operation inherently compositional. Then domains can be extended as shown below. The union of all bindings that apply to message destinations is represented by the relation *dstBinding*.

```
sig Domain {
  ...
  dstBinding: Identifier -> Identifier
}
```

Domain properties

An endpoint is *reachable* in a domain, from an identifier, if there is an address in the closure of the binding relation (but not in its domain) that *routes* to that agent. In this sense, two agents are linked by a chain of one-step connections between handlers; these one-step connections are called *hops*. The chain ends when the last connection reaches the absorbing endpoint.

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

A domain satisfies the *non-looping* property if all those chains of hops and handlers have finite length. This can be written as a predicate in Alloy as follows:

```
pred NonloopingDomain (d: Domain)
  { no ( ^(d.dstBinding) & iden ) }
```

If all the identifiers in a domain reach at most one endpoint, we say that the domain is *deterministic*. Using the Alloy quantifier `lone` (meaning "at most one"), we get

```
pred DeterministicDomain (d: Domain) {
   all i: Identifier | lone g: Agent | ReachableInDomain(d,i,g) }
```

In [4], Zave stated that these domain properties are preserved under addition of bindings, provided that some conditions on the arguments are added. This operation is modelled by the predicate:

```
 AddBinding(d, d' : Domain, newBinding: Identifier -> Identifier),
```

which establishes that `d'` is the result of adding the binding `newBinding` to `d`.

Ensuring the preservation of reachability only requires to state that all the identifiers appearing in the domain of *newBinding* are neither in the domain of *routing*, nor in the domain, nor in the range of the old *dstBinding*. These conditions are formalized by the predicate `IdentifiersUnused`. The following assertion states the preservation condition explained before.

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

To preserve determinism it is sufficient to have `IdentifiersUnused(d, newBinding.Identifier)` and know that *newBinding* is deterministic. To preserve non-looping it is necessary to have `IdentifiersUnused(d, newBinding. Identifier)` and know that *newBinding* is non-looping.

In [4], Zave used the Alloy Analyzer to show that no counterexample for these preservation assertions exists in models up to certain sizes. Using DPS we proved that all of them hold for models of any size.

Returnability

Binding is essential to create persistent network connections between endpoints. This is done by delivering messages between them. In this model, to send a message destined to the generator of a previously received one is called to *return* a message. A domain satisfies the *returnability* property if every return message is delivered to the corresponding agent. This property is stated by the predicate `ReturnableDomain` in the Alloy specification.

When a message travels across handlers to reach an endpoint, its source identifier can be modified. This identifier is used by the receiver as the destination of the return message. Then, when a binding is added to a domain it is necessary to inform whether that binding will change those identifiers. In this model, this is done by the specialization of the `AddBinding` operation into two new operations: `AddABinding` (no source modification) and `AddBBinding`.

As mentioned in [4], it is possible to ensure returnability in domains, such as those defined in this model, by imposing certain conditions on them. These conditions are: bindings and routing operate on different identifiers; except for *A bindings*, delivering a message is deterministic; *B bindings* are invertible; and *A bindings* precede *B bindings*. A domain satisfying those conditions is called a *structured* domain.

Using the Alloy Analyzer, Zave showed in [4] that structure guarantees returnability within the bounds used for the analysis. Using DPS we proved that the addition of any of the two kinds of bindings preserves the structure of a domain (as determined by the predicate `StructuredDomain`).

4 Using DPS to prove properties in an Alloy model

Among the various proof commands provided by PVS, there are some that are critical in the development of a proof because their application could determine whether it is possible to close a proof. Those commands perform the following actions: *hiding of sequent formulas*, *introducing lemmas* and *introducing cases*.

The DPS proof commands implementing these actions have shown to be very useful in two ways. The first one was already mentioned and refers to the fact that whenever one of these actions is taken the correctness of the formula being introduced is analyzed in order to avoid the introduction of non-valid hypothesis. The second one is that DPS helps in the construction of the proof by *pruning of goals*.

To illustrate the impact of these proof commands we will use parts of the proof of the assertion `BindingPreservesReachability` mentioned before.

4.1 Introduction of cases

As we already mentioned, introduction of cases splits a branch in two, or more, sub-goals. When a branch is split, one must solve all of the newly produced branches to close it. At that moment, one would like to have certain confidence on the fact that all the new branches can be closed, otherwise it will not be possible to complete the proof.

As an example. let us see the simplified proof tree shown in Fig. 1. This tree schematically represents the original proof tree. The marked nodes are those in which a *case* command was applied. Notice that those nodes are at almost every level in the proof and are the main reason why a branch splitting may occur. Notice that a mistake in the introduction of a case can invalidate a significant part of the proof.

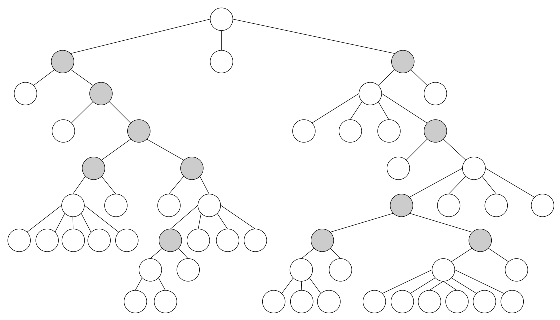

Fig. 1. Simplified proof tree of the assertion `BindingPreservesReachability`.

It was at that point where DPS became helpful by enabling the possibility of using the Alloy Analyzer on the new subgoals, in order to search for counterexamples of the respective sequents. Many working hours were saved by avoiding the introduction of errors by case splitting.

4.2 Hiding sequent formulas

A formula can become irrelevant at some point of a proof for many reasons. For instance, if it has to be used in another sibling but not in the current branch. Figure 2 shows how, after the application of some proof commands, the resulting sequent gets very difficult to be understood.

```
{-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. 2. Example of a sequent after the application of some proof commands.

Notice that only two formulas are relevant (`1` and `-1`). This is not an unusual situation, because predicates often are used to encapsulate related properties that not always are used at the same time, and their expansion can easily fill the screen.

As we showed before, irrelevant sequent formulas obfuscate the job of proving a theorem. For that reason, hiding sequent formulas could be a handy feature of any interactive theorem prover like DPS.

Obviously, some formulas are needed to close a branch. Then, if by mistake one of these formulas is hidden, the possibility of completing the proof can be compromised. The DPS prover command `dps-validate-goal` allows the user to search for counterexamples of the logical implication induced by the current proof goal. Internally this is implemented by a call to the Alloy Analyzer on the assertion resulting from the reverse translation of the current sequent.

In this way, the user can (up to a certain bound) check the safety of executing an error prone action like hiding formulas.

4.3 Introduction of validated lemmas

The use of lemmas in the construction of a complex proof is essential. It allows to encapsulate parts of the problem being solved, and to reuse it across the proof. The DPS counterpart of the `lemma` PVS command, allows the user to introduce assertions and/or facts as new hypothesis in the sequent.

It is worth noting that introducing a lemma is a crucial point of the proof because from that point on, the work heavily depends on the validity of that lemma. Using the DPS capability of validating assertions gives the user the chance to start the phase of proving a lemma once the usefulness of the lemma was confirmed (by helping to close the branch in which is introduced).

4.4 Pruning of goals

As mentioned before, it usually happens that a proof goal contains irrelevant formulas. It is very useful to have a mechanism to figure out which ones of the sequent formulas are really useful, specially when the person who wrote the assertion is not the one who has to prove it.

DPS provides such mechanism. When the user applies the proof command `prune-goal`, the system builds the logical implication between the antecedents and the consequents, but keeping one of the formulas apart. Once this is done the resulting assertion is analyzed in order to search for a counterexample. This is done for all of the formulas in the sequent. If one of these searches succeeds (a counterexample is found), it means that the missing formula is relevant, and the user should use it to close the branch. In the other case, the missing formula is deemed to be irrelevant, and the fact that no counterexample was found (up to the given bounds) should encourage the user to think of a proof that does not make any use of that formula. This mechanism is based on the same idea that uses the automatic theorem prover SRASS [11] to select relevant premises of a theorem from a large set of axioms.

This capability has been one of the most useful DPS features in building proofs for the case study. This is because difficult assertions produce complicated sequents that are hard to understand at first glance, and these checks provide a good guidance in the development of the proof. The relevant formulas of the sequent showed in Fig. 2 were discovered by using this command.

5 Discussion

The proof of the Alloy assertions took 21 days of work of a PhD student. A total of 41 lemmas had to be proved. Statistics for each assertion are given in table 1. Note that the sum of the number of lemmas used for each assertion is above 41 because of the reuse of some lemmas.

Table 1. Distribution of the workload

Assertion	Lemmas	Time (days)
`BindingPreservesReturnability`	17	4
`BindingPreservesDeterminism`	30	14
`BindingPreservesNonLooping`	13	1
`ABindingPreservesStructure`	12	1
`BBindingPreservesStructure`	12	1

All the introduced hypothesis were previously validated by the Alloy Analyzer. All the main branches of the proofs where pruned with the help of the Alloy Analyzer. In our experience these were key features for the understanding of the model and helped us save days of work. This can be seen reflected in the following: proving the hardest assertion in this case study took 14 days, while proving an assertion of comparable difficulty in the case study presented in [3] took 23 days to the same person.

In the process of verifying the Alloy specification using Dynamite we realized that the integration between model-checking and theorem proving is a lot more than a position. We experienced a dramatic reduction of the time needed to prove properties based on the fact that whenever we started to prove properties, the comprehension we had about the specification was shallow and most of the time lemmas and case splitting were based on (possibly) incorrect intuitions. The use of the Allow analyzer in the validation of these lemmas and cases revealed severe misunderstandings of the behavior of the model, exemplified by the counterexamples produced after the analysis, thus contributing in getting a better comprehension of the model. With the help of the analyzer, the learning process suffered a dramatic improvement because counterexamples served as guides on how to understand the model behavior, thus helping in future introduction of lemmas and case splittings.

With complex specifications, sequents tend to grow a lot while the proofs are being developed. This happens as a consequence of the appearance of new assumptions introduced by case splitting or the application of certain proof rules.

In some cases these assumptions appear very close to the start of the proof and are maintained throughout all the branches, even when some of these assumptions are useless for many of the branches. The possibility of pruning the sequents by hiding useless formulas provided good guidelines of how the proof should be completed.

We believe that this integration of a model-checker with a theorem prover changes the whole concept of machine-assisted theorem proving by reducing the technical requirements needed in order to use it in industrial software development.

Integration of SAT-solvers and interactive theorem provers has been proposed in several works. In particular, the semi-automatic theorem prover Isabell/HOL [12] allows the use of external tools (model checkers, for example) to demonstrate proof subgoals. An even more cohesive interaction is proposed in [13], where MiniSAT and zChaff SAT-solvers are used to construct proofs of propositional tautologies for Isabell/HOL or to find a counter example in case the proposition is not valid. But in Dynamite the model checker is used to help the user to construct the proof of a property on possible infinite-sized models. The combination of Alloy with a theorem prover was addressed in [14], where the Prioni theorem prover allows to prove properties on Alloy specifications by translating them to first order formulas that describes their semantic. Then, an interactive first-order theorem prover can be used in order to prove the desired property. Nevertheless, the switching between formalisms overloads the work of the user, that have to master a different language, with complete different semantics, in order to be able to prove the Alloy assertions.

6 Further Work

Many ideas emerged from the intrinsic complexity of proving properties. We believe a machine can do much more in assisting users in the verification process. On the one hand there are many concepts applied in programming environments like Eclipse SDK [10] that can be included as features available in the process of proving properties. Some of these concepts are refactoring of proofs (the possibility of renaming resources, in this case predicates, functions and lemmas, or modifying their arity), tracing dependencies of proofs to have a global view of the verification process state, shared proof pieces detection and automatic factorization as lemmas, suggestions of measures to be taken at a certain step of a proof for a certain formula of the sequent, this includes no only proof rules that could unify with that formula, but also the suggestion of lemmas stored in a database.

On the other hand we will work on the possibility of using the model-checker in an implicit way. This means that there is no need for the user to wait for the model-checker as far as it can run in background and report any counterexample it could find as a consequence of a lemma introduction or a case splitting. This approach also enables the possibility of using the analyzer for automatic, and continuous, sequent pruning while the computer is idle.

We believe this kind of features are the ones that will really make theorem provers usable in realistic software development scenarios.

References

1. Jackson, D., Shlyakhter, I., Sridharan, M.: 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, Vienna, Austria, Association for the Computer Machinery, ACM Press (2001) 62–73
2. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In Kapur, D., ed.: Proceedings of the 11th. International Conference on Automated Deduction (CADE). Volume 607 of Lecture Notes in Artificial Intelligence., Saratoga, NY, Springer-Verlag (1992) 148–752
3. Frias, M.F., Lopez Pombo, C.G., Moscato, M.M.: 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 (2007) 587–601
4. Zave, P.: Compositional binding in network domains. In Misra, J., Nipkow, T., Sekerinski, E., eds.: FM. Volume 4085 of Lecture Notes in Computer Science., Springer (2006) 332–347
5. Free software foundation: Gnu emacs. On-line (1998–2008) Available at `http://www.gnu.org/software/emacs/`.
6. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.: PVS prover guide. Computer Science Laboratory, SRI International. Version 2.4 edn. (2001)
7. Frias, M.F., Lopez Pombo, C.G., Aguirre, N.M.: An equational calculus for Alloy. In Davies, J., Schulte, W., Barnett, M., eds.: Proceedings of the 6th. International conference on formal engineering methods (ICFEM). Volume 3308 of Lecture Notes in Computer Science., Seattle, Washington, United States, Springer-Verlag (2004) 162–175
8. Frias, M.F.: Fork algebras in algebra, logic and computer science. Volume 2 of Advances in logic. World Scientific Publishing Co., Singapore (2002)
9. Zave, P.: A formal model of addressing for interoperating networks. In Fitzgerald, J., Hayes, I.J., Tarlecki, A., eds.: FM. Volume 3582 of Lecture Notes in Computer Science., Springer (2005) 318–333
10. The Eclipse foundation: Eclipse sdk. On-line (2001–2008) Available at `http://www.eclipse.org`.
11. Sutcliffe, G., Puzis, Y.: SRASS - A Semantic Relevance Axiom Selection System. In Pfenning, F., ed.: FM. Volume 4603 of Lecture Notes In Artificial Intelligence., Springer-Verlag (2007) 295–310
12. Paulson, L.: Isabelle: A generic theorem prover. In volume 828 of Lecture Notes in Computer Science, Springer (1994)
13. Weber, T.: Efficiently Checking Propositional Resolution Proofs in Isabelle/HOL. In Benzmüller , C., Fischer, B., Sutcliffe, G., ed.: Proceedings of The 6th International Workshop on the Implementation of Logics. CEUR-WS.org/Vol-2 (2006) 44–62
14. Arkoudas K., Khurshid S., Marinov D. and Rinard M.: Integrating Model Checking and Theorem Proving for Relational Reasoning, in Proceedings of RelMiCS 2003 (Relational Methods in Computer Science), LNCS, Springer, 2003.