Electronic Communications of the EASST
Volume 48 (2011)

# Proceedings of the Fifth International Workshop on Foundations and Techniques for Open source Software Certification (OpernCert 2011)

## Formal verification of a theory of packages

Jaap Boender

9 pages

ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

# Formal verification of a theory of packages

**Jaap Boender**[1*]

[1] boender@pps.jussieu.fr
Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126 CNRS, F-75205 Paris, France

**Abstract:** Over the years, open source distributions have become increasingly large and complex—as an example, the latest Debian distribution contains almost 30 000 packages.

Consequently, the tools that deal with these distribution have also become more and more complex. Furthermore, to deal with increasing distribution sizes optimisation has become more important as well.

To make sure that correctness is not sacrificed for complexity and optimisation, it is important to verify the underlying assumptions formally.

In this paper, we present an example of such a verification: a formalisation in Coq of a theory of packages and their interdependencies.

**Keywords:** verification, theorem proving, open source distributions

## 1 Introduction

During the last decade, open source distributions have become more and more popular, and more and more complex. Where the first release of Debian only had 128 packages, the latest is well on its way to 30 000.

In order to deal with these complexities, the MANCOOSI project has developed new tools and methods to help distribution editors gain insight into the structure of their distributions, to more easily discover errors and to help with administrative procedures (such as migrating packages from unstable to stable distributions).

Since distributions are so large and complex, it is very important that these tools be as fast as possible, without sacrificing correctness. We have therefore used the Coq theorem prover to formally verify some of the assumptions used for optimising the MANCOOSI tools.

In the rest of this article, we will present a brief overview of the formalisation and explain the design decisions taken.

## 2 Formalisation

In this section, we will discuss the formalisation of the theory of packages as described in [MBD+06], [ADBZ09] and [DB10]. We shall not repeat the definitions presented there unless absolutely necessary, so that we can concentrate on the formalisation itself.

[*] Partially supported by the European Community's 7th Framework Programme (FP7/2007-2013), grant agreement n°214898, "Mancoosi" project. Work developed at IRILL.

The Coq sources of the formalisation are available through the Web at http://www.pps.jussieu.fr/~boender/package-theory.tar.gz

## 2.1 Definitions

The formalisation has been done in the Coq proof assistant, which comes with an extensive library. The first choice to be made is of how to represent packages and repositories. In the original definition, packages are represented as a pair $(u, v)$, where $u$ is the name and $v \in \mathbb{N}$ the version.

For the formalisation, we have decided to treat packages as atomic entities, and forget the version number completely. This is possible because the theory we want to formalise never uses the specific fact that a package can exist in multiple versions. In fact, the only point where this is used in the real world is in the Debian distribution, where there is an implicit conflict between two packages that have the same name but a different version (in RPM, this is not the case). This specific instance can easily be modelled by adding the implicit conflict ourselves where necessary.

A repository, then, is a set of packages. There are at least three ways in Coq to represent sets: the `ListSet` library, which uses linked lists, the `Ensemble` library, which uses characteristic functions, and the `MSet` library. We have chosen the latter, because it is very modular, flexible, and comes with a large set of theorems. We do not need the complexity of the `Ensemble` library, since package repositories are finite.

The modularity of `MSet` comes to the fore when looking at the definition of a package. The only thing we need to do is to define a package module type. This type is defined as a refinement of the built-in `OrderedType`, with a few standard axioms defining the equality and order relations, as well as a comparison function:

```
Module Type PACKAGE <: OrderedType.
  Parameter t: Set.

  Parameter eq: t -> t -> Prop.
  Parameter lt: t -> t -> Prop.

  Axiom eq_refl: forall x: t, eq x x.
  Axiom eq_sym: forall x y: t, eq x y -> eq y x.
  Axiom eq_trans: forall x y z: t, eq x y -> eq y z -> eq x z.
  Instance eq_equiv : Equivalence eq :=
    Build_Equivalence t eq eq_refl eq_sym eq_trans.

  Axiom lt_trans: forall x y z: t, lt x y -> lt y z -> lt x z.
  Axiom lt_not_eq: forall x y: t, lt x y -> ~ eq x y.
  Axiom lt_strorder: StrictOrder lt.
  Axiom lt_compat : Proper (eq==>eq==>iff) lt.

  Parameter compare: t -> t -> comparison.
  Parameter compare_spec : forall s s', CompSpec eq lt s s' (compare s s').

  Parameter eq_dec: forall x y, { eq x y } + { ~ eq x y }.
End PACKAGE.
```

```
Declare Module Package: PACKAGE.
```

The declaration of a set of packages (i.e. a repository) now can be done very easily:

```
Declare Module PackageSet : MSetInterface.Sets with Module E := Package.
```

All theorems about sets that are part of MSet are now defined for package sets.

The modularity of MSet also makes it very easy to define conflicts. A conflict is defined as a pair of two packages, such that conflicts are symmetrical, and a package never conflicts with itself. We can define this using the built-in type PairOrderedType. This is a functor that, given two OrderedTypes, returns the type of a pair of these two OrderedTypes.

```
Module Conflict := PairOrderedType Package Package.

Declare Module ConflictSet : MSetInterface.Sets with Module E := Conflict.

Axiom conflicts_sym: forall (C: ConflictSet.t) (p q: Package.t),
  ConflictSet.In (p, q) C -> ConflictSet.In (q, p) C.

Axiom conflicts_irrefl: forall (C: ConflictSet.t) (p: Package.t),
  ~ ConflictSet.In (p, p) C.
```

In only four lines, we have now defined conflicts, sets of conflicts, and their properties.

We now have the basic definitions of packages, repositories and conflicts. In the following sections, we will use these definitions to formalise the rest of the theory of packages.

## 2.2 Dependencies

All definitions pertaining to dependencies have been put in a separate module. This is not only for ease of reference, but it also allows us to use the dependency module as a parameter later on. This is especially interesting when we start formalising the dependency cone (the transitive closure of the dependency function, i.e. all packages that could possibly be needed to install another package); by making this independent of the dependency function, we can easily specify different cones: the normal dependency cone, but also the cone that includes only conjunctive dependencies (dependencies that can only be satisfied by a single package), or even the reverse dependency cone (the cone of packages that depend on a given package).

The dependency function itself is modelled as a Coq function
Dependencies : Package.t → listPackageSet.t. The dependencies of a package are specified as a list of alternatives; every alternative is a set of packages, one of which has to be installed for the package itself to be installed.

We have added a 'filter' to the formalisation of dependencies, a function
dep_filter : PackageSet.t → bool. This function allows us to consider only select dependencies, for example conjunctive ones (a dependency is conjunctive iff its set is a singleton). An example of how this works can be seen in subsection 2.4. By default, the dep_filter function simply returns true for any argument, so that all dependencies are taken into account.

The predicate direct_dependency that notes whether a package $q$ is a direct dependency of $p$ (that is to say directly mentioned in the dependencies of $p$):

```
Definition direct_dependency (p: Package.t) (q: Package.t) :=
  exists d: PackageSet.t, In q d ∧
  List.In d (List.filter dep_filter (Dependencies p)).
```

We use the direct dependency predicate to define the dependency relation; a package *p* depends on a package *q* if it is possible to trace a path of successive direct dependencies from *p* to *q*.

```
Function dependency_path (p q: Package.t) (l: list Package.t)
  { struct l }: Prop :=
  match l with
  | nil => direct_dependency p q
  | h::t => direct_dependency p h ∧ dependency_path h q t
  end.

Definition dependency (p q: Package.t): Prop :=
  exists l: list (Package.t), dependency_path p q l.
```

The `dependency_path` function is recursive on *l*; we need to specify this explicitly so that Coq can be sure that the function terminates (the recursive call must be on a subterm of the original argument).

Since the `dep_filter` and `Dependencies` functions are defined as parameters in the dependency module, every function outside the dependency module that calls them will have them added as parameter. This allows us to define functions on other dependency relations, as we will see in subsection 2.4.

## 2.3 The dependency cone

The dependency cone is defined as the transitive closure of the aforementioned dependency function. It is formalised as follows:

```
Function cone (P: {x: PackageSet.t | x [<=] R})
  {measure (fun x => cardinal R - cardinal (proj1_sig P)) P}: PackageSet.t :=
  if equal (inter R (dependencies (proj1_sig P))) (proj1_sig P)
  then (proj1_sig P)
  else
    cone (exist (fun v => v [<=] R) (inter R (dependencies (proj1_sig P)))
    (fun a => inter_subset_1 (s:=R) (s’:=dependencies (proj1_sig P)) (a:=a))).
```

This definition basically states that the cone of a set *P* is the fixed point of the repeated application of the dependency function. There is a lot of extra information, because `Coq` needs to be sure that the function actually terminates.

To start with, the argument *P* is not a simple package set, but a subset of a repository *R* (as stated in the dependent type). This is used for the termination: the `measure` keyword states that the number $|R| - |\texttt{dependencies}\ P|$ strictly decreases. Coq now generates the necessary proof obligations for this statement automatically.

The only disadvantage here is that we can no longer use *P* by itself, since it is not a simple set anymore, but a set with a specific property, and a witness that proves that the set actually has that property. To extract the set, we use the function `proj1_sig`; there is also `proj2_sig` if we need to extract the witness.

Note that for the recursive application of the `cone` function, we have to provide a proof that the recursive argument is also a subset of $R$ (to satisfy the dependent type); we do this using the built-in `MSet` theorem `inter_subset_1` theorem which states that $\forall_{s,s'}[s \cap s' \subseteq s]$.

We can also see that the `cone` function calls the `dependencies` function; if we want to have a cone of another type dependencies, we can specify another dependency function, for example `conjunctive_dependencies` for a cone of only conjunctive dependencies.

## 2.4 Strong dependencies

In [ADBZ09], we introduced the notion of *strong dependencies*; a package $p$ strongly depends on another package $q$ if every installation of $p$ also contains $q$. However, if $p$ itself is not installable, it is discounted, because then 'every installation of $p$' would be the empty set, and so $p$ would strongly depend on every other package in the distribution.

In Coq, this is formalised as follows:

```
Definition satisfied_pkg (S: PackageSet.t) (p: Package.t): Prop :=
  forall d: PackageSet.t, List.In d (Dependencies p) ->
  exists p': Package.t, In p' (inter S d).

Definition abundant (S: PackageSet.t): Prop :=
  PackageSet.For_all (satisfied_pkg S) S.

Definition concerned (S: PackageSet.t) (c: Package.t * Package.t): Prop :=
  let (p, q) := c in (In p S) ∧ (In q S).

Definition peaceful (S: PackageSet.t) (C: ConflictSet.t): Prop :=
  ConflictSet.For_all (fun c => ~(concerned S c)) C.

Definition healthy (S: PackageSet.t) (C: ConflictSet.t): Prop :=
  abundant S ∧ peaceful S C.

Definition is_install_set (p: Package.t) (R: PackageSet.t) (C: ConflictSet.t)
  (I: PackageSet.t) :=
  I [<=] R ∧ In p I ∧ healthy I C.

Definition strong_dep (R: PackageSet.t) (C: ConflictSet.t)
  (p: Package.t) (q: Package.t) :=
  (exists N: PackageSet.t, is_install_set p R C N) ∧
  forall I: PackageSet.t, I [<=] R -> In p I ∧ healthy I C -> In q I.
```

The notions of abundance and peace are explained in [MBD$^+$06]. Informally, a set of packages is abundant iff all the dependencies of its constituent packages are satisfied, and it is peaceful iff there are no conflicting packages within the set. If a healthy (abundant and peaceful) set can be found that contains a package $p$, then $p$ is installable, and the set is called an install set of $p$.

As for the definition of strong dependency, it is necessary that $p$ is installable, as noted above, and every healthy set that includes $p$ must also include $q$.

We can use this definition to prove an optimisation of the algorithm to find strong dependencies (as presented in [ADBZ09]). This optimisation uses the fact that a conjunctive dependency is automatically a strong dependency (a conjunctive dependency only has one alternative, which means that this alternative must necessarily be installed). We can therefore mark all conjunctive dependencies as strong dependencies, which drastically reduces the search space.

In order to formalise this notion, we first define the notion of a conjunctive dependency:

```
Definition conjunctive_dependency (R: PackageSet.t) (p q: Package.t): Prop :=
  dependency Dependencies is_conjunctive_bool p q.
```

The `conjunctive_dependency` function is simply defined as the dependency function with another dependency filter. Now we see why the definition of `Dependencies` and `dep_filter` as parameters is useful: we parametrise the conjunctive dependency function with the normal `Dependencies` function, but we filter them with `is_conjunctive_bool`, which only returns `true` if its argument is a conjunctive dependency.

Now we can use this `conjunctive_dependency` function to define the theorem:

```
Theorem conjunctive_strong_dep: forall R C p q,
  conjunctive_dependency R p q -> installable R C p ->
  strong_dep R C p q.
```

## 2.5 Strong conflicts

In [DB10], we introduced the notion of strong conflicts, i.e. pairs of packages that cannot be installed together under any circumstances. We also presented an algorithm to quickly find all the strong conflicts in a given distribution.

This algorithm is much faster than the naive method of just checking every possible pair, by using a theorem that states that for two packages $p$ and $q$ not to be installable together, there must be an explicit conflict $(c, c')$ such that $p$ depends on $c$ and $q$ depends on $c'$. The algorithm can then traverse the dependency tree backwards from every explicit conflicts, which limits the number of candidates to check.

This theorem is easy to formalise. First the definition of a strong conflict:

```
Definition strong_conflict (R: PackageSet.t) (C: ConflictSet.t)
  (p: Package.t) (q: Package.t) :=
  installable R C p ∧ installable R C q ∧
  ~exists I: PackageSet.t, healthy I C ∧ In p I ∧ In q I.
```

The definition of the theorem:

```
Theorem scp: forall (R: PackageSet.t) (C: ConflictSet.t)
  (p: Package.t | In p R) (q: Package.t | In q R),
  strong_conflict R C (proj1_sig p) (proj1_sig q) ->
  ConflictSet.Exists (fun c =>
     match c with
       (c1,c2) => (E.eq (proj1_sig p) c1 ∨ normal_dependency (proj1_sig p) c1)
                ∧ (E.eq (proj1_sig q) c2 ∨ normal_dependency (proj1_sig q) c2)
     end) C.
```

The proof can be found in [DB10]; the Coq proof closely follows this scheme.

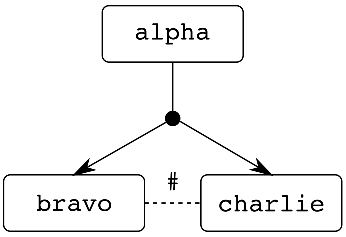

Figure 1: Example of a triangle conflict

## 2.6 Triangle conflicts

An optimisation to the strong conflict algorithm uses the concept of *triangle conflicts*. Here is its definition:

**Definition 1** A conflict $(c_1, c_2)$ is a triangle conflict if and only if there exists a package $p$ such that:

- there is a $d \in \texttt{Dependencies}(p)$ such that $c_1 \in d$ and $c_2 \in d$;
- there is no other $p'$ such that $p'$ depends on either $c_1$ or $c_2$.

An example of this situation can be found in figure 1; `bravo` and `charlie` are in conflict, and there are no packages that depend on them except for `alpha`.

We have proven that triangle conflicts can be discarded when checking for strong conflicts. Informally, the reason for this is that since the only way to get to the two packages in conflict ($c_1$ and $c_2$) is through $p$; in order to install $p$, we can choose either $c_1$ or $c_2$. Since there are no other packages that depend on $c_1$ or $c_2$, which one we choose does not matter.

In Coq, we express this by the following theorem:

```
Theorem triangles_ok:
  forall (R: PackageSet.t) (C: ConflictSet.t) (a: PackageSet.t | a [<=] R),
  (For_all (fun a => installable (NoSupDependencies R) R C a) (proj1_sig a)) ->
  only_triangles R C (fun _ => true) ->
  ~ConflictSet.Exists (fun c => let (c1, c2) := c in
    In c1 (proj1_sig a) ∨ In c2 (proj1_sig a)) C ->
  co_installable (NoSupDependencies R) R C (proj1_sig a).
```

This expresses that given a repository $R$, a set of conflicts $C$, and a set of packages $a$, if all packages in $a$ are installable separately, if the only conflicts are triangle conflicts, and if there is no direct conflict between any members of $a$, then all packages from $a$ must be installable together.

The proof sketch is as follows:

- By induction on $a$.

  1. If $a = \emptyset$, then all packages in $a$ are trivially installable together.

2. If $a = a_1 \cup \{x\}$, then:
   - By the induction hypothesis, all packages from $a_1$ are installable together (take $A_1$ as the installation set). Also, $x$ is a member of $a$, and therefore it is installable by the original hypothesis (with $A_x$ as the installation set).
   - Now, we take the union of $A_1$ and $A_x$, where we remove $c_1$ or $c_2$ if both are present. This set is an installation set as well (it contains all packages needed and no conflicts).

- Hence, all packages from $a$ are installable together.

## 3 Conclusion and discussion

We have presented a short overview of the formalisation of the theory of packages as used in the MANCOOSI project. More details can be found in the Coq files, which are available through the Web (see the previous section for the exact location).

As for future work, the formalisation is not yet complete. For example, the proof presented in [Boe11] has not yet been formalised, mostly due to the lack of an appropriate graph theory library in Coq, especially one that integrates well with `MSet`.

After this extension, the logical next step would be to verify the actual algorithms used in the tools; these use the theorems presented above, but only proving the theorems is no guarantee for correctness.

There are several methods to execute this verification; it could be done using the `Program` extension to Coq [Soz07], which automatically generates proof obligations for programs written in Coq, based on invariants and assertions supplied by the user.

Another method would be code extraction, where the code is first written in Coq, proved correct, and then ‘extracted’ to OCaml. The resulting code would be faster than the code generated by Coq and `Program`, but if there were to be a bug in the code extraction process, it could produce false results.

Most of the algorithms in MANCOOSI rely on a SAT solver to check for installability. Formally verifying a modern SAT solver would be very difficult, but it might be possible to treat the SAT solver as a black box and only verify its result.

**Acknowledgements:** The author would like to thank Roberto Di Cosmo and Pierre Letouzey for their very useful advice and encouragement.

## References

[ADBZ09] P. Abate, R. Di Cosmo, J. Boender, S. Zacchiroli. Strong dependencies between software components. In *ESEM ’09: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. Pp. 89–99. IEEE Computer Society, Washington, DC, USA, 2009.
doi:http://dx.doi.org/10.1109/ESEM.2009.5316017

[Boe11] J. Boender. Efficient Computation of Dominance in Component Systems. In *Proceedings of SEFM*. 2011.

[DB10] R. Di Cosmo, J. Boender. Using strong conflicts to detect quality issues in component-based complex systems. In *ISEC '10: Proceedings of the 3rd India software engineering conference*. Pp. 163–172. ACM, New York, NY, USA, 2010. doi:http://doi.acm.org/10.1145/1730874.1730905

[MBD+06] F. Mancinelli, J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, X. Leroy, R. Treinen. Managing the Complexity of Large Free and Open Source Package-Bas ed Software Distributions. In *ASE*. Pp. 199–208. 2006.

[Soz07] M. Sozeau. Subset Coercions in Coq. In Altenkirch and McBride (eds.), *Types for Proofs and Programs*. Lecture Notes in Computer Science 4502, pp. 237–252. Springer Berlin / Heidelberg, 2007.