Dependency Management for the Eclipse Ecosystem: An Update

Daniel Le Berre[1]* Pascal Rapicault[2]

[1] Univ Lille Nord de France, F-59000 Lille, France
UArtois, CRIL, F-62307 Lens, France
CNRS, UMR 8188, F-62307 Lens, France
leberre@cril.univ-artois.fr
[2] Sonatype
Mountain View, CA 94040
rapicault@sonatype.com

Abstract. One of the strength of Eclipse, the well-known open platform for software development, is its extensibility made possible by the built-in pluggability mechanisms. However, those pluggability mechanisms only reveal their full potential when extensions created by others are made easy to distribute and obtain. The purpose of Eclipse p2 project is to build a platform addressing the challenges of distribution and obtention of Eclipse and its extensions, which poses the same dependency management issues as for component based systems. This paper focuses on the dependency management aspect of p2. It describes a boolean optimization encoding of the dependency management problem of Eclipse. In particular, it focusses on the changes made since the initial adoption of such approach in Eclipse two years ago. We conclude by lessons learned while using propositional logic to model a real world problem.

1 Introduction

Eclipse[3] is a very popular open platform mainly written in Java and designed from the ground up as an integration platform for software development tools but also for rich client applications [16]. As the Eclipse ecosystem becomes more and more important, the Eclipse platform itself and the vertical platforms built on Eclipse all rely on the concept of extensibility, and as such the necessity for a mechanism to acquire those extensions is primordial. To that end, almost since its inception, Eclipse featured an extension acquisition mechanism named Update Manager. However, over time, as inter plug-in dependencies became more complex and expressed at a finer grain, and more versions of each component were made available, limitations were being discovered in Update Manager which

* Part of this work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the' Contrat de Projets Etat Region (CPER) 2007-2013'. The work on explanations detailed in section 3.3 has been supported in part by Genuitec company.

[3] http://www.eclipse.org/

were hindering the adoption and retention of Eclipse. The term "plug-in hell" was coined. It is at that time that we started to work on Eclipse p2 with the goal of building a "right-grained" provisioning platform attempting to address the challenges that Update Manager had been faced with.

The first challenge was heterogeneity in the set of things being deployed, since it had become clear over time that most OSGi- and Eclipse-based applications needed to have a manageable way to interact with their environment (e.g JRE, Windows registry keys, etc.).

The second challenge was the need to address in one platform the diversity of provisioning scenarios and offer a solution that would work against controlled repositories -similar to the case of linux packages managed by a specific Linux distribution- or uncontrolled repositories, would allow for fully automated solutions or user-driven ones, or would sport the delivery of extensions as well as complete products.

The final and most important challenge was to solve the "plug-in hell", i.e., the difficulty for the end user to install a plug-in and its requirements. That problem was partially rooted in the non modular way of acquiring components used in the Update Manager: it forced extensions to be installed by a special abstraction one level above the actual extension itself. The term "right-grained" provisioning is a response to this problem and indicates that p2 is not an obstruction to the granularity of what a user would want to make available or obtain.

In order to achieve this goal of "right-grained" provisioning, the efficiency, reliability and scalability of the dependency resolver was key. Having learnt from our experience of authoring the OSGi runtime resolver for Equinox, it was obvious that we would need to base our dependency analysis mechanism on proven solver techniques. Coincidentally, later that year, the work of OPIUM[19] and EDOS[14] backed up our intuition on the usability and maturity of a SAT-based approach to address the problem. Our main contribution here compared to that existing work is to deal with the more complex dependencies of Eclipse and to have built a solution that is currently running on millions of computers. The dependency problem for Eclipse is closer to the problem addressed by the follow-up to EDOS project, the Mancoosi Project[1, 18], that is the problem of updating complex open source environments. All the details concerning Eclipse metadata and the original translation of Eclipse dependency problems into pseudo boolean optimization problems can be found in [12]. The adaptation of that approach in the context of Linux dependencies and the Mancoosi project is presented in [4]. In this paper we present the details of the new implementation that will ship with Eclipse 3.6 in June 2010. Finally, we conclude with lessons learned while using propositional logic to model a real world problem.

2 p2 metadata

The concept of metadata is at the core of most installers that deal with composable systems (e.g RPM, Debian, etc.). One of the goals of this metadata,

and the point of focus of this paper, is to capture the dependencies that exist between the components of the system. A resolver uses that metadata to find missing dependencies or to validate the dependencies of the system before it is modified. As described previously, p2 is intended to deal with more than just the typical Eclipse constructs of OSGi bundles. As such, despite the presence of dependency information in the OSGi bundles composing most of Eclipse applications, p2 abstracts dependencies from the elements being delivered in an entity called an *Installable Unit* (also referred to as *IU*). We now introduce the two most widely used kinds of installable units that p2 defines.

2.1 Anatomy of an installable unit

An installable unit, the simplest construct, has the following attributes:

An identifier A string naming the installable unit.

A version The version of the installable unit. The combination identifier and version is treated like a unique ID. We will refer to versions of an installable unit to mean a set of installable units sharing the same identifier but a different version attribute.

A set of capabilities A capability is the way for the installable unit to expose to the rest of the world what it has to offer. This is just a namespace, a name and a version. Namespace and name are strings. The namespace is here to prevent name collision and avoid having everyone adhere to name mangling conventions.

A set of requirements A conjunction of requirements. A requirement is the way for the IU to express its needs. Requirements are satisfied by capabilities. A requirement is composed of a namespace, a name and a version range [4]. In addition to these usual concepts, a requirement can have a filter (under the form of an LDAP filter [9]) which allows for its enablement or disablement depending on the environment where the IU will be installed, and it can also be marked optional meaning that failing to satisfy the requirement does not prevent the IU from being installable. Finally there is a concept of greed discussed later in this section.

An enablement filter An enablement filter indicates in which contexts an installable unit can be installed. Here again the filter will pass or fail depending on the environment in which the IU will be installed.

A singleton flag This flag, when set to true, will prevent a system from containing another version of the installable unit with the same identifier.

An update descriptor The identifier and a version range identifying predecessors to this IU. Making this relationship explicit allows us to deal with IUs being renamed or avoid undesirable update paths.

[4] A version range is expressed by two version number separated by a comma, and surrounded by an angle bracket, meaning value included, or a parenthesis, meaning value excluded.

An example of an Installable unit representing the SWT bundle is given in Figure 1. The few things to notice are the usage of namespace to avoid clashes between the Java packages and the IU identifier; the usage of singleton because no two versions of this bundle can be installed in the same eclipse instance; the "typing" of the IU as being a bundle (see namespace `org.eclipse.equinox.p2.type` valued to `bundle`); and the identification of the IU by providing a capability in the `org.eclipse.equinox.p2.iu` namespace.

```
id=org.eclipse.swt, version=3.5.0, singleton=true
Capabilities:
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.swt, version=3.5.0}
    {namespace=org.eclipse.equinox.p2.eclipse.type name=bundle version=1.0.0}
    {namespace=java.package, name=org.eclipse.swt.graphics, version=1.0.0}
    {namespace=java.package, name=org.eclipse.swt.layout, version=1.2.0}
Requirements:
    {namespace=java.package, name=org.eclipse.swt.accessibility2,
       range=[1.0.0,2.0.0), optional=true, filter=(&(os=linux))}
    {namespace=java.package, name=org.mozilla.xpcom,
       range=[1.0.0, 1.1.0), optional=true, greed=false}
Updates:
    {namespace=org.eclipse.equinox.p2.iu, name=org.eclipse.swt,
       range=[0.0.0, 3.5.0)}
```

Fig. 1. An IU representing the SWT bundle

Now, let's come back to requirements and detail the semantics of greed and optional. By default, a requirement is "strong"[5] (optional is false, greed is true). This means that the IU can only be installed if the requirement is met. If a "strong" requirement is guarded by a filter that does not pass, the requirement is ignored. When the optional flag is set to true, then a requirement becomes "weak" and it does not have to be satisfied for the IU to be installed. However, any IU potentially satisfying this requirement will be considered, and a best effort will be made to satisfy the requirement.

When it comes to greed, this is a rather atypical concept that we have added to control the addition of IUs as part of the potential IUs to install in order to satisfy the user request. When the greed is true (default case, and the case for strong requirements), the IUs satisfying the dependencies are added to the pool of potential candidates. However, when the greed is set to false, such a requirement relies on other dependencies from its own IU or others to bring in what is necessary for its satisfaction. This is used in Figure 1 to capture the fact that even though we have an optional dependency on `org.mozilla.xpcom` we don't want to try to satisfy it eagerly. As such, this optional and non greedy

[5] Strong is weaker in our context than the notion of strong dependency introduced in [3]

requirement is weaker than a typical optional dependency. Table 1 reviews the four combinations of greed and optionality.

Greed	Optional	Semantics
true	false	this is a "strong" requirement.
true	true	this is a "weak" requirement.
false	true	this is a "weakest" requirement, where the match will not be brought in.
false	false	this indicates a case where the requirement has to be satisfied but the IU with this requirement wants this to be brought in by another one.

Table 1. Greed and optional interaction.

2.2 Installable unit patch

The need for patches So far, the concept of IU is pretty much on par with what most package managers are offering. However, what is interesting is the different usage we have observed of this metadata and the implication it has on the rest of the system. Indeed, most people building on top of Eclipse are delivering "products" or "subsystems", and, as such, they want to guarantee that their customer is getting what has been tested. Failing to do this could result in a non stable product, maintenance nightmare and unsatisfied customers. However, in an ecosystem where products can be mixed and where repositories can not be used as control points [6], guaranteeing a functional system is harder. Consequently, to palliate these possible problems, product producers are using installable units as a grouping mechanism (also referred to as *group*) serving three goals:

1. Facilitate the reusability of a set of functionality by aggregating under one group a set of installable units.
2. Capture a particular configuration of the system, and thus group under one IU an extensible element and a default implementation.
3. Lock down the dependencies on installable units being used, which limits the variability of what can be installable and thus guarantees reproducibility of an installation independently of the content of the repository.

The counter part of the lock down which is used extensively throughout Eclipse, is that it makes the delivery of service (e.g., the replacement of a particular IU by another one) complex for the following reasons:

1. Products are often made of groups, themselves recursively composed of other groups, which can make for a rather vast ripple effect throughout the system when a low level component needs to be serviced.

[6] Controlled repository is the approach taken by a majority of linux distributions.

2. Not all groups deployed on the user's machine are in the control of the same organization. For example, someone can be running a composition of Sonatype and Artois University products (both including the Eclipse Platform group), but the Platform group is controlled by the Eclipse open source community. Therefore when the Platform team needs to deliver a fix to a user, it simply can not require all the referring groups to be updated.
3. Not all the dependencies on a particular IU are known ahead of time.

2.3 Overview of the solving process

Before detailing the overall solver, it is worth mentioning how p2 manages the installed software. p2 has a concept of *profile* which keeps track of two key pieces of information: the list of all the Installable Units installed, and the set of *root installable units*. The root IUs are not a new kind of installable units, they are installable units that are remembered as having been explicitly asked for installation. These roots are essential for installation, uninstallation and update, since they are used as strict constraints that can't be violated, thus for example avoiding the uninstallation of an IU when installing another one.

p2 resolution process is logically organized in 5 phases:

Change request processing Given a *change request* capturing the desire to install or uninstall an installable unit, a future root set representing the application of this request over the initial root set is produced.

Slicing For each element in the future root set, the slicing produces a transitive closure of all the IUs (referred to as *slice*) that could potentially be part of the final solution of the resolution process by consulting all repositories also passed in. This transitive closure is done with only taking into account enough context[7] to evaluate the various filters but without worrying if any IU being added could be colliding with any others.

Projection/encoding The goal of the projection phase is to transform all the installable units of the slice and their dependencies into a pseudo boolean optimization problem (see section 3 for details).

PBO-solving The result of the projection is passed to the pseudo boolean solver Sat4j[11] which is responsible for finding an assignment.

Solution extraction From the assignment returned by the solver, a solution is extracted. In case of failure, the solver is invoked to produce an explanation (see section 3.3).

3 Constraints encoding

In the following, we describe the encoding of the p2 installation problem into propositional constraints, i.e., clauses or cardinality constraints. We also provide some examples of problems generated with that encoding. In the following, IU_x^v

[7] The context can be seen as a map of key/value pair

will denote the installable unit x in version v. We will use the same notation to represent the propositional variables. We will simply write IU_x when no information is provided for the version. $prov(IU_x)$ denotes the set of capabilities provided by the installable unit IU_x and $req(IU_x)$ denotes the set of capabilities required by the installable unit IU_x. $alt(cap) = \{IU_k | cap \in prov(IU_k)\}$ denotes the set of IUs providing a given capability cap. Finally, $optReq(IU_x)$ denotes the optional requirements of a given IU_x, and $versions(IU_x)$ denotes the ordered set of IUs sharing the same identifier as IU_x but having different version attribute ($IU_x \in versions(IU_x)$), from the latest to the oldest.

3.1 Basic encoding

Each requirement of the form "IU_i requires capability cap_j" is represented by a simple binary (Horn) clause

$$IU_i \rightarrow cap_j$$

So, for each IU_i the requirements are expressed by a conjunction of binary clauses

$$\bigwedge_{cap_j \in req(IU_i)} IU_i \rightarrow cap_j$$

The alternatives for a given capability is given by the clause

$$cap_j \rightarrow IU_{j_1}^{v_{j_1}} \vee IU_{j_2}^{v_{j_2}} \vee ... \vee IU_{j_n}^{v_{j_n}}$$

where $IU_x^{v_x} \in alt(cap_j)$.

Since we are only interested in the IUs to install, the above two constraints can be aggregated into a conjunction of constraints:

$$f(IU_i) = \bigwedge_{cap_j \in req(IU_i),} (IU_i \rightarrow \bigvee_{IU_x^v \in alt(cap_j)} IU_x^v) \quad (1)$$

Note that there is the specific case of $alt(cap_j) = \emptyset$ which means that IU_i cannot be installed due to missing requirements. In that case, the unit clause $\neg IU_i$ is generated.

Some installation units cannot be installed together (e.g., because of the singleton attribute set to true). This can be modeled either with a conjunction of binary negative clauses

$$\bigwedge_{versions(IU_x)=<IU_x^{v_1},...,IU_x^{v_n}>,1 \leq i < j \leq n} (\neg IU_x^{v_i} \vee \neg IU_x^{v_j})$$

or equivalently with a single cardinality constraint:

$$(\sum_{IU_x^{v_j} \in versions(IU_x)} IU_x^{v_j}) \leq 1 \quad (2)$$

We use the second option because our solver manages those constraints natively and because it makes the explanation support easier to implement (see 3.3 for details).

Finally, the user wants to install the installable units identified by the roots. This is modeled with unit clauses:

$$\bigwedge_{UI_i^j \in rootIUs} UI_i^j \tag{3}$$

Summing up, the constraints (1), (2) and (3) together form an instance of a classical NP- complete SAT problem. This encoding is basically the encoding presented in Edos[14] and Opium [19] and used more recently in OpenSuse 11[8].

3.2 Eclipse specific encoding

One of the specificity of p2 is the semantic of "weak" dependencies expressed using the `greed` and `optional` attributes.

Encoding of optionality An IU IU_i may have optional dependencies to IU IU_j meaning that IU_j is not mandatory to use IU_i, so IU_i can be installed successfully if IU_j is not available. However, it is expected that p2 should favor the installation of optional packages if possible, i.e., that all optional packages that could be installed are indeed installed. In Figure 1, one can see that SWT has two optional dependencies on SWT accessibility2 and Mozilla XPCOM. The encoding of optional packages is done by creating two specific propositional variables: Abs_{cap} denotes the fact the capability *cap* is optional, and $Noop_{IU_i}$ is a variable to be satisfied in case none of the optional capabilities of IU_i can be installed. The first set of constraints expresses how to satisfy the required capabilities:

$$\bigwedge_{cap_j \in optReq(IU_i)} (Abs_{cap_j} \rightarrow \bigvee_{IU_x \in alt(cap_j)} IU_x) \tag{4}$$

The main issue now is to allow the formula to satisfy all, some or no capabilities without violating any constraint. Our initial encoding, and the one that is still shipping with Eclipse, was based on a disjunction of the capabilities with an additional *Noop* variable plus a linear objective function. However, the right answer to such problem is to use a non linear optimization function: Each Abs_x variable gets a reward to favor the installation of optional dependencies when the requiring package IU_j is installed: $\sum -K \times Abs_{cap_i} \times IU_j$. $-K$ is the reward for installing both IU_j and its optional requirement cap_i.

Example 1. Let's see how to encode the optional dependencies of SWT on accessibility2 and xpcom shown in Figure 1:
$Abs_{accessibility2} \rightarrow IU^{1.0}_{accessibility2}$

[8] `http://en.opensuse.org/Package_Management/Sat_Solver/Basics`

$Abs_{xpcom} \rightarrow IU^{1.1}_{xpcom}$
$min : -K \times Abs_{acessibility2} \times IU_{SWT} - K \times Abs_{xpcom} \times IU_{SWT}$

Encoding of non greedy requirements In the original encoding, the non greedy requirements were simply managed during the slicing stage and ignored in the resolving stage. However, it appeared that in order to allow a finer control of non-greedy requirements, it was better to let the resolver manage those requirements. The encoding of non greedy requirements is based on the introduction of new propositional variables NG_X that are satisfied iff IU X is provided by a greedy requirement. Each requirement of the form "IU_i requires non greedily capability cap_j" is encoded the following way:

$$f(IU_i) = \bigwedge_{cap_j \in reqNonGreedy(IU_i),} (IU_i \rightarrow \bigvee_{IU^v_x \in alt(cap_j)} NG\_U^v_x) \quad (5)$$

Then, the non greedy IUs are associated to the IUs that require them greedily:

$$NG\_U^v_x \rightarrow \bigvee_{IU^v_x \in alt(cap_j), cap_j \in req(IU^j_i)} U^j_i \quad (6)$$

One can note that such encoding will favor the installation of IUs providing non greedy requirements. In case no such IUs are found, the NG_X variables will be set to false, thus falsifying equation 5.

A similar approach is used for optional non greedy dependencies, by introducing NG_X variables in equation 4.

Encoding of patches Applying a patch from the encoding point of view only applies to requirements changes (see section 2.2), i.e., it means to enable or disable some dependencies according to the application or not of a given patch. Adding or removing capabilities or requirements is not implemented yet, but does not bring any difficulty from an encoding point of view. We denote by $patchedReqs(IU, p)$ the set of pairs $< old, new >$ of the installable unit IU denoting the rewriting rules of patch p in the requirements of IUs.

We associate to each patch a new propositional variable. We introduce that variable in dependency constraints (1) and (4) the following way:

- Negatively to express the new dependency brought by the patch.
- Positively to express the initial dependency. In that case, all patches changing that dependency should appear positively in the constraints: if none of them are applicable, the initial dependency is applied.

It can be summarized in this way:

$$\bigwedge_{<old,new> \in patchedReqs(IU,p)} (\neg p \vee encode(new)) \wedge (\bigvee_{<old,new_i> \in patchedReqs(IU,p_i)} p_i \vee encode(old))$$

where $encode(x)$ denote the encoding of a regular or an optional dependency. The patch encoding changes only the encoding of the requirements affected by a patch.

3.3 When things go wrong: explanation

Explanation is key helping the user understand why a change request cannot be fulfilled. In the above encoding, one can note that there are only two reasons that could prevent a request from succeeding:

- At least one of the required IUs is missing.
- The request requires two IUs sharing the same identifier but with different versions that cannot be installed together due to the singleton attribute on at least one of those IUs.

As a consequence, it is not hard to check why a request cannot be completed. However, users expect the explanation to be returned in terms of IUs they know about, the root IUs and the IUs that they are trying to install, and would be confused if provided with just the low level dependency errors. In practice, it means that knowing why a problem occurred is not sufficient. It is important to be able to detail the whole dependencies from the root to the actual cause of the problem.

Let S be the set of the constraints encoding presented in the previous sections. From a logical point of view, it is possible to compute one minimal subset S' of the constraints that cannot be satisfied altogether: $S' \subseteq S, S' \models \bot, \nexists S'' \subset S' | S'' \models \bot$. Such set of constraints is often called a MUS (minimal unsatisfiable subformula). S' is an explanation of the impossibility to fulfill the request. If the subset contains a negated literal (specific case of Equation (1), $\neg UI_x \in S'$) then the global explanation is a missing requirement, i.e., the request cannot be completed because IU_x cannot be found in the user's repositories. If the subset contains a cardinality constraint ($\sum IU_v^x \leq 1 \in S'$), then the global explanation is a singleton attribute violation, i.e., the request cannot be completed because it requires several versions of IU_v. Note that if we decided to use a clausal encoding instead of the cardinality constraints encoding, we would have lost the one to one mapping between the original dependencies and the constraints of our encoding.

There are several ways in practice to compute S' from S. The ones based on local search algorithms[17] detect constraints that are likely to be part of S' among the most falsified ones during the search and compute S' in a second step using a complete SAT solver. A more recent and widely used approach is based on the analysis of the last conflict found by a conflict driven SAT solver[20]. Such approach requires some changes in the SAT solver to keep track of all resolutions steps and does not ensure that the computed subset $S'' \subseteq S$ is minimal. A third approach is to rely on a new encoding of the problem into an optimization problem using selector variables [13]: it is possible to use an optimization function on selector variables to compute a set S' of minimal size. Finally, a generic approach to explanation in constraints solvers was proposed in [10] and implemented in Ilog solver: QuickXplain. The main advantage of such approach is that it is independent of the underlying solver, and that it works with any kind of constraints.

Our approach inherits some ideas from all those approaches. We decided to implement the QuickXplain algorithm in our framework because it is non intrusive (does not require any change to the solver) and works perfectly with mixed

constraints (clauses and cardinality constraints in our case). We use selector variables in our encoding to allow the QuickXplain algorithm to enable/disable the constraints when computing S'. Finally, we restrict the number of selector variables to be considered by the QuickXplain algorithm using a specific conflict analysis procedure in the spirit of [6].

More precisely, we translate S into S'' by adding a new selector variable sel_i to each constraint in S: $S'' = \{sel_i \vee s_i | s_i \in S\}$. Let SEL denotes the set of all added selector variables. Instead of looking for an assignment satisfying S, we are looking for an assignment satisfying S'' under the assumption that all variables in SEL are set to false, $S'' \bigwedge_{sel_i \in SEL} \neg sel_i$[9]. If such assignment exists, it is an assignment satisfying S, so we are done. If it is not the case, then we analyze the last, top level conflict found by the SAT solver: we derive from that analysis a subset of the selector variables that caused the inconsistency. We use a tailored version of the QuickXplain algorithm that makes use of that subset of selector variables to enable/disable constraints in order to compute S'.

The reduction of the selector variables to consider in the QuickXplain algorithm allowed us to reduce some explanation time from tens of seconds to a few seconds. It has been introduced when we discovered that the explanation process in huge development repositories (many different versions of all the installable units released by the Eclipse foundation) could take up to one minute on a recent quad-core computer.

3.4 From decision to optimisation

When all the constraints can be satisfied, there are usually many possible solutions, that are not of equal quality for the end user. Here are a few remarks regarding the quality of the expected solution:

1. An IU should not be installed if there is no dependency to it.
2. If several versions of the same bundle exist, the latest one should preferably be used.
3. When optional requirements exist, the optional requirements should be satisfied as much as possible.
4. User installed patches should be applied independently of the consequences of its application (i.e., the version of the IUs forced, the number of installable optional dependencies, etc.).
5. Updating an existing installation should not change packages unrelated from the request being made.

We are now looking for the "best" solution, not just any solution, i.e., we moved from providing a certificate for the answer to a decision problem (NP-complete from a complexity theory point of view) to return the solution of an optimization problem (NP-hard). Furthermore, we need to solve a multi-criteria optimization problem since it is likely that several IUs do have optional requirements and

[9] Assumption based satisfiability testing is available in all Minisat[8] inspired solvers (including Sat4j).

that several IUs are available in multiple versions. The optimization criteria we are dealing with here are much more complex than the ones presented earlier in [19].

To solve our problem, we build a linear optimization function to minimize in which the propositional variables are either given a penalty (positive integer) or a reward (negative integer) to prevent or favor their appearance in the computed assignment.

- Already installed packaged and Root Installable Units should be kept installed whenever possible. However, it should be possible to update the packages found in the transitive closure of the requirements of the Root IUs:

$$\sum_{IU_v^i \in (Installed \setminus transitiveClosure(Root)) \cup Root} 1 \times IU_v^i \tag{7}$$

- Each version of an IU gets a penalty as a power of $P = max(|Installed|+1, 2)$ proportional to its age, the older it is the more penalized it is:

$$\sum_{IU_v^i \in versions(IU_v) \setminus (Installed \cup Root)} P^i \times IU_v^i \tag{8}$$

  That way, each installation of an IU raises a penalty at least by one, thus expressing that only required IUs should be installed.
- We have seen that we need to add a non linear combination of boolean variables in our objective function for managing optional dependencies: $\sum -P^{K+1} \times Abs_{cap_i} \times IU_j$. The problem is that our solver does not propose yet an easy way to work with non linear optimization functions. A solution based on the introduction of new variables fixes that issue:

$$\sum -P^{K+1} \times y_k \text{ with } y_k \leftrightarrow Abs_{cap_i} \times IU_j \tag{9}$$

- Each *patch* variable gets a reward of $n \times -P^{K+3}$ if it is applicable (where n denotes the number of applicable patches), else a penalty of P^{K+2}

$$\sum_{p_i \in applicablePatches()} n \times -P^{K+3} p_i + \sum_{p_i \notin applicablePatches()} P^{K+2} p_i \tag{10}$$

The objective function of our optimization problem is thus to minimize (7) + (8) + (9) + (10).

The weights in (8) are not satisfactory since they do not provide a total order on the final solution. Suppose that we have two IUs, IU_a and IU_b, that are available in 3 and 2 versions, respectively (namely IU_a^3, IU_a^2, IU_a^1 and IU_b^2, IU_b^1) with $P = 2$. The objective function for those IUs is thus

$$2 \times IU_a^3 + 4 \times IU_a^2 + 8 \times IU_a^1 + 2 \times IU_b^2 + 4 \times IU_b^1$$

The best solution for such objective function if both IU_a and IU_b must be installed is obviously to install IU_a^3 and IU_b^2. However, if those two IUs cannot

be installed together, the solver will answer that the best option is either to install IU_a^3 and IU_b^1 or IU_a^2 and IU_b^2.

The common approach to solve this problem is to rank each IUs in a total order, $IU_1 < IU_2 < ... < IU_m$, meaning that IU_i is more important than IU_j iff $IU_j < IU_i$. Then the coefficients of the optimization function should be generated in such a way that the sum of the coefficients of IU_j should be smaller than the smallest coefficient of IU_i. In our example, it would mean for instance to use the following optimization function:

$$IU_a^3 + 2 \times IU_a^2 + 4 \times IU_a^1 + 8 \times IU_b^2 + 16 \times IU_b^1$$

In that case, the best option is still to install IU_a^3 and IU_b^2, but the second best option is to install IU_a^2 and IU_b^2.

Unfortunately, as noted before, we are in the context of uncontrolled repositories, so there is no obvious/easy way to order the IUs in a total order, so it was decided to keep the initial solution (8) instead of arbitrarily ranking the IUs.

Equation (7) has been introduced at the users' request. Indeed, some "stability" is needed for vendors building their tools on top of the Eclipse platform, for quality assurance for instance. The idea of keeping as much of possible the already installed packages was designed for that reason. However, in the open source world, it is often desired that installing a new software also updates its dependencies. This is the reason why the installable units found in the transitive closure of the requirements of the Root IUs are not "glued" to their installed version.

4 Conclusion and perspective

We presented the boolean optimization encoding used in Eclipse p2, a "right-grained" provisioning platform aimed at solving the diversity of provisioning requirements in a componentized world. The initial encoding has evolved over the years to include both user's feedback and modeling improvements.

We can report that this approach has been live for two years now, and that is has been used by millions of users worldwide[10]. It has proven to be reliable, efficient and scalable even when faced with repositories containing more than 10000 installable units and solution involving about 3000 installable units. Furthermore, a direct consequence of our work is the integration of the very same technology to manage dependencies in other Java related products, namely the upcoming major release of Maven[11] (Maven3) and the repository manager Nexus.

The feasibility tests done in late 2007 to investigate the use of SAT technology inside the Eclipse platform revealed that most of the dependencies could be resolved with few backtracking. The main issue was not to find a solution, but to find an expected solution, i.e., to correctly model the computation as a boolean

[10] `http://www.eclipse.org/downloads/`

[11] `http://maven.apache.org/`

optimization problem the expected behavior of the resolver. The initial solution based on the Pseudo-Boolean solver as a black box was not satisfactory from a modeling point of view: using the common input format defined for the Pseudo Boolean evaluations is not user friendly, especially for software engineers. The introduction last year of a tighter integration with the Sat4j library allowed to model directly the constraints on Java objects, which proved to be much easier to improve the boolean optimization encoding, and allowed several developers to experiment with their own optimization criteria.

While the size of the repositories available in the Eclipse world is compatible with our current implementation, scaling is an issue for the future. We recently used a similar approach to resolve some Linux upgradeability problems proposed by the Mancoosi European project[4]. Those problems are basically one order of magnitude bigger than the Eclipse ones (up to 50K packages). While adapting our work to the Linux world allowed us to quickly provide a correct tooling for solving those problems, some classes of problems were really challenging to our implementation. Replacing Sat4j by another optimization engine (namely the MaxSAT solver MsUnCore[15]) proved to be more efficient, but some problems remain challenging. The same scaling issue will appear in Maven world, where the size of the central repository is around 200K packages! Furthermore, the dependency management is contextualized in that case to the state of the build process (compile time, runtime, etc). As such, a modular, maybe even user oriented, way of defining what is a good (optimal) solution is needed.

Acknowledgements

The authors would like to thank the anonymous reviewers' valuable comments and suggestions on the improvement of this article.

References

1. Mancoosi, Managing the Complexity of the Open Source Infrastructure. http://www.mancoosi.org.
2. OSGi Service Platform. http://www.osgi.org/Specifications.
3. Pietro Abate, Jaap Boender, Roberto Di Cosmo, and Stefano Zacchiroli. Strong dependencies between software components. Technical Report 2, Mancoosi - Seventh Framework Programme, May 2009.
4. Josep Argelich, Daniel Le Berre, Ines Lynce, Pascal Rapicault and Joao Marques-Silva. Solving Linux Upgradeability Problems Using Boolean Optimization. In *Proceedings of LoCoCo2010 - Workshop on Logics for Component Configuration*, 2010.
5. Josep Argelich, Ines Lynce, and Joao Marques-Silva. On solving boolean multi-level optimization problems. In *Twenty-First International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 393–398, Pasadena, California, USA, 2009.
6. Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Efficient generation of unsatisfiability proofs and cores in sat. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *LPAR*, volume 5330 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 2008.

7. Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. A simple and flexible way of computing small unsatisfiable cores in sat modulo theories. In Joao Marques-Silva and Karem A. Sakallah, editors, *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 334–339. Springer, 2007.
8. Niklas Eén Niklas Sörensson. An extensible sat-solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing, LNCS 2919*, pages 502–518, 2003.
9. IETF. http://www.ietf.org/rfc/rfc2254.txt.
10. Ulrich Junker. Quickxplain: Preferred explanations and relaxations for over-constrained problems. In Deborah L. McGuinness and George Ferguson, editors, *AAAI*, pages 167–172. AAAI Press / The MIT Press, 2004.
11. Daniel Le Berre and Anne Parrain. Sat4j, a SATisfiability library for java. http://www.sat4j.org.
12. Daniel Le Berre and Pascal Rapicault. Dependency Management for the Eclipse Ecosystem. Proceedings of IWOCE2009 - Open Component Ecosystems International Workshop, August 2009.
13. Ines Lynce and Joao P. Marques Silva. On computing minimum unsatisfiable cores. In *SAT*, 2004.
14. Fabio Mancinelli, Jaap Boender, Roberto di Cosmo, Jérôme Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the complexity of large free and open source package-based software distributions. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE06)*, pages 199–208, Tokyo, JAPAN, september 2006. IEEE Computer Society Press.
15. Vasco Manquinho, Joao Marques-Silva, and Jordi Planes. Algorithms for Weighted Boolean Optimization. In *Proceedings of SAT'09*, pp. 495–508, 2009.
16. Mark Powell (NASA) Marc Hoffmann, Gilles J. Iachelini (CSC). Eclipse on rails and rockets. http://live.eclipse.org/node/750.
17. Bertrand Mazure, Lakhdar Sais, and Eric Gregoire. Detecting logical inconsistencies. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics(AI/Math'96)*, pages 116–121, Fort Lauderdale (FL-USA), jan 1996.
18. Ralph Treinen and Stefano Zacchiroli. Solving package dependencies : from Edos to Mancoosi. In *DebConf'8*, Argentine, 2008.
19. Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. Opium: Optimal package install/uninstall manager. In *ICSE*, pages 178–188. IEEE Computer Society, 2007.
20. Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable boolean formulas. In *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT03)*, 2003.