Boolean-Based Dependency Management for the Eclipse Ecosystem

Daniel Le Berre, Pascal Rapicault

Boolean-based Dependency Management for the Eclipse Ecosystem

Daniel Le Berre
CRIL-CNRS UMR 8188, Université d'Artois

Pascal Rapicault
Rapicorp, inc.

Abstract

In June 2008, the Eclipse open platform released a new dependency management system called p2. That system was based on the translation of the dependency management problem into a pseudo-Boolean optimization problem, to be handled by the Sat4j solver. Since then, p2 has been more tightly integrated with Sat4j, the platform opened a public plugin repository (the Eclipse marketplace) which relies on p2 to install the available plugins and their dependencies, and became the favorite way to install plugins in the Eclipse community. This paper summarizes the issues raised by Eclipse dependency management, its pseudo-Boolean encoding within p2, its extension for Linux package management with p2cudf, and concludes with lessons learned on using research software in production systems.

1 Introduction

Eclipse[1] 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[31]. As the Eclipse ecosystem becomes more and more important, the Eclipse platform itself and the various tools 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 discovered in *Update Manager* which were hindering the adoption and retention of Eclipse. Since Software Dependency Management was shown NP-Complete[32, 26], this should not come as a surprise. However, incomplete approaches have been used successfully since the very beginning for Linux package management with tools like `dpkg` and `rpm`,

[1] `http://www.eclipse.org/`

and their limitations only showed up when the number of available packages became extremely large [34, 26]. The term "plug-in hell" was used in the Eclipse community to express the difficulty to manage a product based on plugins line-up. It was 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. We built p2 on top of SAT technologies.

The first challenge was heterogeneity in the set of things being deployed, since it had become clear over time that most OSGi-[2] 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 scenarii 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 were key requirements. 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 reliable solver techniques. Coincidentally, later that year, the work of OPIUM[34] and EDOS[26] backed up our intuition on the usability and maturity of a SAT-based approach to address the problem. The main contribution in p2 compared to that existing work was to deal with the more complex dependencies of Eclipse and to have built a solution that is currently running in production 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, 33], that is the problem of updating complex open source environments. The original translation of Eclipse dependency problems into pseudo-Boolean optimization problems found in the tool `p2` was described during IWOCE 2009 workshop[23]. The adaptation of p2 in the context of Linux dependencies and the Mancoosi project in the tool `p2cudf` was presented during LoCoCo 2010 workshop[8]. In this paper we present the concepts behind those tools and the details of the implementation of `p2` which is used in Eclipse since

[2]Open Service Gateway initiative is a standard component architecture in Java, used in Eclipse since release 3.0.

release 3.6 in June 2010. The Eclipse platform has been downloaded more that 180 million of times from Eclipse web site[3] since the first release of p2 in June 2008. Finally, we conclude with lessons learned while using research software in production systems.

2 Preliminaries on Boolean decision and optimisation problems

In this paper, we model our problems with constraints over Boolean variables V. Boolean values are interpreted as 0 (false) or 1 (true) in this context. A variable is satisfied if it is assigned value 1 and falsified if assigned value 0. A literal is a Boolean variable (v) or is negation ($\neg v$). A literal is satisfied if it represents a satisfied variable or the negation of a falsified variable, else it is falsified. We consider in this paper two types of constraints: clauses and pseudo-Boolean constraints. A clause is a disjunction of literals (e.g., $\neg v_1 \vee \neg v_2 \vee v_3$). A clause $\neg a \vee b$ can also be represented by the implication $a \rightarrow b$, denoting that if a is true then b must be true. We use both notations in this document. A pseudo-Boolean constraint is a constraint of the form $\sum_{i=1}^{n} w_i \times l_i \oplus k$ where l_i are literals, w_i are integers and $\oplus \in \{\leq, \geq\}$, i.e. it is a linear inequality over Boolean variables. A specific case is when all w_i are equal to 1. We call those constraints *cardinality* constraints, since satisfying those constraints amount to count the number of satisfied literal: for instance, $l_1 + l_2 + l_3 + l_4 + l_5 \geq 2$ means that at least two of the five literals must be satisfied. One should note that a cardinality constraint $\sum_{i=1}^{n} l_i \geq 1$ is equivalent to the clause $l_1 \vee l_2 \vee ... \vee l_n$. Thus pseudo-Boolean constraints generalize clauses, so we can assume all our constraints to be pseudo-Boolean constraints.

Given a set of constraints $C = \{c_1, c_2, ..., c_n\}$, we are interested in four related problems.

The first one is deciding if it is possible to find an assignment satisfying all those constraints. If such assignment exists, the set of constraints is said satisfiable (SAT), else it is said unsatisfiable (UNSAT). When C contains only clauses, this problem is called SAT[13], and was shown NP-complete in the general case[15]. Since any Boolean formula can be translated into an equivalent conjunctive normal form (set of clauses), then the complexity of the satisfiability for pseudo-Boolean constraints is still NP-complete. However, finding efficient ways to perform such translation in practice is an active research topic (see e.g., [17, 21] for CNF encoding of cardinality constraints).

The second one applies only when C is found unsatisfiable. In that case, we would like to compute an explanation of the unsatisfiability, i.e. a subset $C' \subseteq C$ of the constraints which is unsatisfiable and for which no proper subset is unsatisfiable. Take for instance $C = \{a \vee b, \neg a, \neg b, a \vee c, \neg a \vee d\}$. $C' = \{a \vee b, \neg a, \neg b\}$ is an unsatisfiable subset of C. All subsets of C' are satisfiable. We will call such C' a minimal unsatisfiable subformula (MUS).

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

The third problem is to find the "best" assignment satisfying those constraints, which means solving the optimization problem

$$\text{minimize } f(V) \text{ given } C$$

where $f(V)$ is a linear fonction over the Boolean variables V. This problem is called pseudo-Boolean Optimization[27] in the SAT community or zero-one linear programming in the Operational Research Community.

The fourth problem is yet another optimisation variant of the first one. In this case, the constraints in C are associated with a weight (an integer) representing the cost to pay if the constraint cannot be satisfied, so $C = \{(w_1, c_1), (w_2, c_2), ..., (w_n, c_n)\}$. We aim at finding an assignment of the variables which maximizes the sum of the weights of satisfied constraints or minimizes the weights of falsified constraints. A specific weight, ∞, denote that the constraint must be satisfied. So the first problem could be written $C = \{(\infty, c_1), (\infty, c_2), ..., (\infty, c_n)\}$ in that context: all constraints must be satisfied. Constraints with weight ∞ are called *hard* while the others are called *soft*. When all the constraints are soft clauses, and the weights are all 1, that problem is called MaxSAT. If some of the clauses are hard, it is called Partial MaxSAT. When all the constraints are soft clauses, without restriction on the weights, the problem is called Weighted MaxSAT. If some of the clauses are hard, it is called Weighted Partial MaxSAT (WPMS)[24].

It is possible to translate a PBO problem as a WPMS problem. Given a linear optimisation function to minimize $f(V) = w_1 \times l_1 + w_2 \times l_2 + ... + w_m \times l_m$ and a set of constraints C, we can build the set of weighted constraints C' such that $C' = \{(\infty, c') | c \in C, c' \in CNF(c)\} \cup \{(w_1, \overline{l_1}), (w_2, \overline{l_2}), ..., (w_n, \overline{l_m})$ where $CNF(c)$ denote a function translating pseudo-Boolean constraints into a set of clauses and $\bar{l}$ denote the opposite literal of l ($\overline{v} = \neg v$ and $\overline{\neg v} = v$).

It is also possible to translate a WPMS problem into a PBO problem. We need to introduce in that case new variables called "selector variables", which will disable the original soft constraint when its selector variable is satisfied. Given a set of weighted clauses C, we build a set of clauses $C' = \{c_i | (\infty, c_i) \in C\} \cup \{s_i \vee c_i | (w_i, c_i) \in C, w_i \neq \infty\}$. All hard clauses are unchanged. Only soft clauses are modified. Let S be the set of the selector variables. The following PBO will answer the original PWMS problem:

$$\text{minimize } \sum_{s_i \in S} w_i \times s_i \text{ given } C'$$

One can note that PBO and WPMS are completely equivalent if the pseudo-Boolean constraints of the PBO problem are just clauses (no need to translate those constraints into CNF) and the weighted soft clauses of the WPMS problem are unit (in that case, we can directly use those unit clauses in the objective

function[22]). Thus when the input is a CNF and an objective function to minimize.

In practice, there exists numerous solvers available for tackling those problems, mainly thanks to the regular SAT[4], MAXSAT[5] or PBO[6] competitions. The efficiency of MUS solvers has greatly improved in recent years in the SAT community[11, 19, 20, 30], especially since the organization of an MUS track in the SAT competition in 2011, i.e. several years after the work presented here.

In this paper, we will use Sat4j[7][22], an open source Java library for solving both decision (like SAT) or optimisation (like PBO and WPMS) Boolean problems. Being written in Java, it's purpose is not to be the fastest solver around but to provide to the end user easily embeddable cutting-edge SAT technology. It is also an experimental research platform for devising new PBO algorithms. Sat4j is one of the few platforms which combine in a single library solvers for tackling all the above problems.

3 Problem input: p2 metadata

The concept of metadata is at the core of most installers that deal with composable systems (e.g RPM[8], Debian[9], etc.). One of the goals of this metadata is to capture the dependencies that exist between the various components of the system. A *resolver* uses that metadata to detect missing dependencies or to validate the dependencies of the system before it is modified: its aim is to ensure that the modified system will work properly.

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.

3.1 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

[4] `http://www.satcompetition.org/`
[5] `http://www.maxsat.udl.cat`
[6] `http://www.cril.univ-artois.fr/PB16/`
[7] `http://www.sat4j.org/`
[8] `http://rpm.org`
[9] `https://www.debian.org/doc/manuals/debian-reference/ch02.en.html`

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[10]. In addition to these usual concepts, a requirement can have a filter 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. Eclipse being multi-platform, the operating system may be used for instance to filter out some IUs.

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.

An example of an Installable Unit representing Eclipse Standard Widget Toolkit (SWT) bundle is given in Figure 1. The few things to notice are:

1. the usage of namespace to avoid clashes between the Java packages and the IU identifier;
2. the usage of singleton because no two versions of this bundle can be installed in the same eclipse instance;
3. the "typing" of the IU as being a bundle (see `org.eclipse.equinox.p2.type` namespace valued to `bundle`);
4. and the identification of the IU by providing a capability in the `org.eclipse.equinox.p2.iu` namespace.

Now, let us come back to requirements and detail the semantics of "greed" and "optional". By default, a requirement is "strong"[11] (`optional` is `false`,

[10] 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.

[11] Strong is weaker in our context than the notion of strong dependency for Linux[7]

```
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)}
```

Figure 1: An IU representing the SWT bundle

Table 1: Greed and optional interaction.

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.

`greed` is `true`). This means that the IU can only be installed if the requirement is met. Note that if a requirement is guarded by a filter that does not pass, that requirement is simply ignored. If 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 do not want to try to satisfy it eagerly. As such, this optional and non greedy requirement is weaker than a typical optional dependency. Table 1 reviews the four combinations of greed and optionality.

3.2 Installable unit patch

3.2.1 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 unstable 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[12], guaranteeing a functional system is harder. Consequently, to mitigate 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 software fixes (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 lead to a rather vast ripple effect throughout the system when a low level component needs to be serviced.

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 Company A and Company B 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.

It is in order to overcome these limitations that we have introduced the concept of IU patch. An IU patch can be seen as a mechanism that rewrite the dependencies of existing IUs and provide its own capabilities which allows for the replacement of IUs without redelivering the whole application.

[12] Controlled repository is the approach taken by a majority of Linux distributions.

3.3 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 already 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, in contrast with installable units which have been installed to satisfy the dependencies of the root IUs. These roots are essential for installation, uninstallation and update, since they are used as strict constraints that can not 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[13] to evaluate the various filters but without worrying if any IU being added could be colliding with any others. That slicing stage proved essential in practice for two reasons: limiting the number of IUs to consider for the next stage to reduce the size of the encoding, and avoiding to update an IU which is not related to the change request. That stage is similar to the "cone of influence" approach used in Bounded Model Checking[12].

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 4 for details).

PBO-solving The result of the projection is passed to the pseudo-Boolean solver Sat4j[22] which is responsible for finding an assignment. Note that for efficiency and reproducibility reasons, the solver does not look for an optimal solution, but for the best solution found within a given number of conflicts (a measure which is computer independent).

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 4.3).

[13] The context can be seen as a map of key/value pairs.

4 Translation into a Pseudo-Boolean Optimization problem

In the following, we describe the encoding of the p2 installation problem into a Pseudo-Boolean Optimization problem, i.e. clauses or cardinality constraints and an objective function. We also provide some examples of problems generated with that encoding. In the following, IU_x^v 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 sequence 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.

4.1 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[14]

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

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 Sat4j[22] manages those constraints natively and because it makes the explanation support easier to implement (see 4.3 for details).

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

$$\bigwedge_{UI_i^j \in rootIUs} UI_i^j \quad (3)$$

Summing up, the constraints (1), (2) and (3) together form an instance of the classical NP- complete SAT problem. This encoding is basically the encoding presented in Edos[26] and Opium [34] and used more recently in OpenSuse 11[15].

4.2 Eclipse specific encoding

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

4.2.1 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 optional capabilities:

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

[14]There are many other more efficient ways to encode the above cardinality constraint with clauses, using additional propositional variables[17, 21].

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

The second set of constraints expresses that if $Noop_{IU_i}$ is true then all the abstract capability variables must be false, i.e. that $Noop_{UI_i}$ can only be set to true when none of the optional dependencies could be installed.

$$\bigwedge_{cap_j \in optReq(IU_i)} (Noop_{IU_i} \rightarrow \neg Abs_{cap_j}) \tag{5}$$

Finally, we express that IU_i has optional dependencies using a disjunction ending with the $Noop_{IU_i}$ variable. That way, even if none of the optional requirements can be installed, the constraint can still be satisfied by setting $Noop_{IU_i}$ to true.

$$IU_i \rightarrow \bigvee_{cap_j \in optReq(IU_i)} Abs_{cap_j} \vee Noop_{IU_i} \tag{6}$$

Satisfying as much as possible the optional requirements can thus be done by minimizing the number of satisfied $Noop$ variables, i.e. by adding those $Noop$ variable in the objective function of our optimization problem.

In practice, it happened that such approach had the consequence to favor the installation of IUs found in the optional requirements of an IU IU_i even if that IU_i was not installed. We solved it using 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 us see how to encode the optional dependencies of SWT on accessibility2 and xpcom shown in Figure 1:*
$Abs_{accessibility2} \rightarrow IU^{1.0}_{accessibility2}$
$Abs_{xpcom} \rightarrow IU^{1.1}_{xpcom}$
$min : -K \times Abs_{acessibility2} \times IU_{SWT} - K \times Abs_{xpcom} \times IU_{SWT}$

Note that Sat4j does not support non linear optimization functions. It relies on the introduction of new variables $y_k \leftrightarrow Abs_x \times IU_j$ where y_k replaces $Abs_x \times IU_j$ in the objective function and where the additional constraints $y_k \rightarrow Abs_x \vee IU_j$, $Abs_x \rightarrow y_k$, $IU_j \rightarrow y_k$ are added to the solver.

Note that a much simpler solution appeared to us, when we finally noticed that the optional requirements are typically soft constraints, like in MaxSAT: one would like to satisfy as many of them of possible. The translation from PWMS to pseudo-Boolean Optimization described in section 2 can be applied here. As such (4), (5) and (6) could simply replaced by

$$g(IU_i) = \bigwedge_{cap_j \in optReq(IU_i),} Noop_{UI_i,cap_j} \vee (IU_i \rightarrow \bigvee_{IU^v_x \in alt(cap_j)} IU^v_x) \tag{7}$$

Let us take the problematic case of imbricated optional dependencies[23]. Suppose that IU_a has an optional dependency on capability b provided by IU_b

that has in turn an optional dependency on the capability c provided by IU_c for which we have $alt(c) = \emptyset$. We would generate the following constraints:

$Noop_{IU_a,b} \vee (IU_a \rightarrow IU_b) \equiv Noop_{IU_a,b} \vee \neg IU_a \vee IU_b$
$Noop_{IU_b,c} \vee (IU_b \rightarrow \emptyset) \equiv Noop_{IU_b,c} \vee \neg IU_b$

In that case, the resolver has the possibility to install or not IU_b, depending of the value of the objective function.

4.2.2 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:

$$f(IU_i) = \bigwedge_{cap_j \in reqNonGreedy(IU_i),} (IU_i \rightarrow \bigvee_{IU_x^v \in alt(cap_j)} NG\_U_x^v) \quad (8)$$

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

$$NG\_U_x^v \rightarrow \bigvee_{IU_x^v \in alt(cap_j), cap_j \in req(IU_i^j)} U_i^j \quad (9)$$

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 8.

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

4.2.3 Encoding of patches

Applying a patch from the encoding point of view only applies to requirements changes (see section 3.2), 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 in p2, but it 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.

4.3 When things go wrong: explanation

Explanation is key in 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. Let S' be an MUS of S. 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.

4.4 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. Furthermore, we need to solve a multi-criteria optimization problem since it is likely that several IUs do have optional requirements and that several IUs are available in multiple versions. The optimization criteria we are dealing with here are much more complex than the ones presented for Linux with Opium[34].

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:

$$\sum_{IU_v^i \in (Installed \setminus transitiveClosure(Root)) \cup Root} 1 \times IU_v^i \tag{10}$$

- 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:

$$\sum_{IU_v^i \in versions(IU_v) \setminus (Installed \cup Root)} P^i \times IU_v^i \tag{11}$$

  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:

$$\sum -P^{K+1} \times y_k \text{ with } y_k \leftrightarrow Abs_{cap_i} \times IU_j \tag{12}$$

- 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}

$$\sum_{p_i \in applicablePatches()} n \times -P^{K+3} p_i + \sum_{p_i \notin applicablePatches()} P^{K+2} p_i \quad (13)$$

The objective function of our optimization problem is thus to minimize (10) + (11) + (12) + (13).

The weights in (11) 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 (11) instead of arbitrarily ranking the IUs.

Equation (10) 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 as 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.

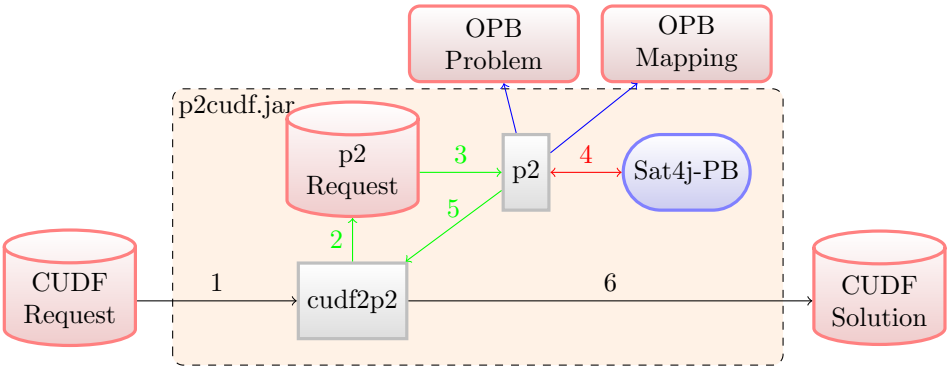

Figure 2: Flow diagram of p2cudf to get a solution from a CUDF instance.

5 p2cudf: From Eclipse to Linux dependency management

The aim of the `p2cudf` resolver[16] was to use the p2 resolver in the context of the Linux distribution to see if our approach was reusable in another context. The Mancoosi project[1], and its Mancoosi International Solver Competition (MISC)[6, 3] has been a great opportunity to compare our approach to other ones[18, 28, 8]. The competition had a common input format called Common Upgradeability Description Format (CUDF). p2 could be reused "as is" since there are just a few differences between CUDF and Eclipse metadata, the main ones being the use of ranges in the version of the packages, and the notion of optional and non greedy dependencies between packages. However, in order to allow the integration of the various criteria defined for the competition[6], and to avoid modifying a software running in production, a simplified version of p2 code was used. The organization of the solver is depicted in Figure 2.

The first step is to translate the CUDF request into a p2 request: each CUDF package is translated into an Installable Unit. Once the translation is finished, the p2 resolver translates the request into a pseudo-Boolean optimization problem. It can either directly feed Sat4j solver or output it in a file with the corresponding mapping. In the former case, the p2 resolver gets a solution from Sat4j. That solution is then translated into a simplified CUDF universe (with only the package name, version and installation status).

The next sections describe more formally the constraints used in p2 and the way the optimization functions have been designed to implement the Mancoosi optimization criteria[6].

[16] `https://wiki.eclipse.org/Equinox/p2/CUDFResolver`

5.1 Original constraints

In the following, p_i^v will denote package p_i in version v. We will use the same notation to represent the propositional variables. We will simply write p_i when no information is provided for the version. $versions(p_i)$ will denote the set of available versions of the package p_i in the universe. $Installed$ and $NonInstalled$ will represent the versions of the packages that are respectively installed or not in the universe.

CUDF constraints are translated as follows:

- p_i^x depends on $p_j^{x_j}, j \in [1..n]$ is translated into a clause $\neg p_i^x \vee p_1^{x_1} \vee ... \vee p_n^{x_n}$
- p_i^x conflicts with $p_j^{x_j}$ $(i \neq j)$ is translated into the binary clause $\neg p_i^x \vee \neg p_j^{x_j}$
- In case a specific package p_i^x conflicts with p_i, which means that no more than one version of the same package can be installed, we use a cardinality constraint $\sum_{x \in versions(p_i)} p_i^x \leq 1$ which is natively supported in Sat4j[22].
- A package p_i^v that cannot be found in the universe will be denoted by the negative literal $\neg p_i^v$
- A package p_i^v that is requested to be installed or updated will be denoted by the positive literal p_i^v

5.2 Handling inconsistency of installed packages

One of the big differences between Eclipse metadata and Linux metadata is that an Eclipse profile (the current installation) is always consistent with the metadata while a Linux installation might not be consistent with the metadata (the user may force the installation of a package). This is partly due to the fact that in the Linux world, dependencies are meant to describe what has been validated by QA: they denote preferred configurations, i.e. violating those constraints may still end up with a runnable system. In the Eclipse ecosystem, the dependencies describe the requirements of the classloader: if those dependencies cannot be satisfied then the dependent package will not be activated.

As a consequence, when a CUDF universe denotes a set of installed packages, those packages dependencies might not be satisfied: mapping each installed package p_i^v to the positive literal p_i^v – as it is the case in the Eclipse encoding – would end up in that case with an inconsistent pseudo-Boolean optimization problem.

For that reason, the packages marked as "installed" in the CUDF universe are considered as "optional requirements" in the Eclipse p2 terminology: the solver will try to install as many of them as possible, but will not fail if none of them can be installed.

This is achieved by the following constraints, for which we introduce a $Noop$ propositional variable that will prevent the clause to be falsified if none of op-

tional requirements can be installed:

$$Root \rightarrow p_1^{v_1} \vee p_2^{v_2} \vee ... \vee p_n^{v_n} \vee Noop \text{ for } p_i^{v_i} \in Installed$$

$$Noop \rightarrow \neg p_i^{v_i} \text{ for } i \in [1..n]$$

The fact that a maximum of $p_i^{v_i} \in Installed$ will be installed will be managed by the optimization function.

5.3 Objective functions

The most important part of the translation is to correctly express the criteria used in the competition. We are using here a translation of the lexicographic preference into a single optimization function. This is possible because our pseudo-Boolean engine does support arbitrary precision coefficients. For each measure used in a criterion, we introduce new variables and constraints between those new variables and existing ones in order to express the optimization criteria only on newly introduced variables.

5.3.1 Common definitions

A package is removed in the solution if it was installed but is no longer available in any version. For any package p_i installed in the original CUDF, we introduce a new variable $removed_{p_i}$ such that

$$removed_{p_i} \equiv \bigwedge_{p_i^x \in versions(p_i)} \neg p_i^x, p_i \in Installed$$

A package is changed in the solution if the status of one of its versions (installed or not) has changed. For any package p_i in the original problem, we introduce a new variable $changed_{p_i}$ that is true *iff* the status of any version of p_i has changed:

$$changed_{p_i} \equiv \bigvee_{p_i^x \in Installed \cap Versions(p_i)} \neg p_i^x \bigvee_{p_i^x \in NonIntalled \cap Versions(p_i)} p_i^x, p_i \in U$$

A package is not up to date if that package is installed but the latest version available is not installed. For any package p_i in the original problem, we introduce a new variable $notuptodate_{p_i}$ such that

$$notuptodate_{p_i} \equiv \neg latest(p_i) \wedge (\bigvee_{p_i^x \in versions(p_i) \setminus latest(p_i)} p_i^x), p_i \in U$$

.

A package is new if the package was not installed but appears installed in the solution. For any package p_i that was not installed in the original CUDF, we introduce a new variable new_{p_i} such that

$$new_{p_i} \equiv \bigvee_{p_i^x \in versions(p_i)} p_i^x, p_i \in NonInstalled$$

5.3.2 Paranoid criterion

The paranoid criterion is a lexicographic preference on the number of packages removed and then on the number of changed packages. In that context, our optimization function on $|Installed| + |U|$ variables is

$$(|U| + 1) \times \sum removed_{p_i} + \sum changed_{p_j}$$

5.3.3 Trendy criterion

The trendy criterion is a lexicographic preference on the number of packages removed, the number of packages that are not up-to-date and the number of newly introduced packages. In that context, our optimization function on $2 \times |U|$ variables is

$$(|U|+1) \times (|U|+1) \times \sum removed_{p_i} + (|U|+1) \times \sum notuptodate_{p_j} + \sum new_{p_k}$$

5.4 p2cudf against the other solvers

During the three editions of the MISC competitions (2010, 2011 and 2012), p2cudf has been compared to different other solvers, namely ASPCUD[18] based on the ASP solver CLASP, UNSA[28] based on the MILP solver CPLEX, inesc [8] based on the Maxsat solver MSUNCORE. The complete raw results can be found online[3]. They are also discussed in details[6].

p2cudf performed correctly on medium size instances, but had trouble scaling up when the number of packages becomes large. The best approach during the competition was UNSA, based on CPLEX, a commercial solver. We demonstrated that the experience gained when working on Eclipse dependency management could be helpful in the context of Linux package management. Some features of p2, such as the slicing stage, proved to be quite useful in practice for MISC too.

6 Conclusion and perspectives

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 running for nine years now, and that is has been used by millions of users worldwide. 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. Since 2010, a plugin "market" has been made available `https://marketplace.eclipse.org` to the Eclipse users: more than 26 million of plugins have been installed that way. Eclipse marketplace takes into account plugin dependencies thanks to p2.

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 the expected solution, i.e., to correctly model as a Boolean 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 one year later 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.

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 used a similar approach to resolve some Linux upgradeability problems proposed by the Mancoosi European project within p2cudf. 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.

The integration of Sat4j in Eclipse is a success story: it is probably one of the most significant adoptions of SAT technology in a widely used software product. We believe it happened for two independent but key reasons. First, the Eclipse community decided that the "plugin hell" had to be sorted out, by all means, once for all. The principle to delegate that work to an external tool instead of relying on an in house solver was key toward that goal. Second, Sat4j was developed from the beginning to be used in production software, and designed in the open source spirit. For instance, Sat4j has been developed within the ObjectWeb, now OW2, consortium since 2005. The eligibility of Sat4j as a reliable third party component for the Eclipse platform was simplified by such open development model. Note also that Sat4j was built from the beginning using Eclipse, which motived us to work on this problem.

In this context, Eclipse had a problem and looked for a solution from academia. This has been possible because the development of Eclipse is governed by a central body, the Eclipse Foundation. Linux has the very same issue with its package management, but its distributed nature prevented so far a similar solution to be adopted[4]. A solution to that problem has been proposed as outcome of the Mancoosi European Project: the Mancoosi Package Manager[5]. Unfortunately, this package manager is not currently used by default on most Linux distributions. New generations of package managers do appear. Dandified Yum (DNF)[17] for instance is SAT-based, but relies on a tailored SAT-solver, not an off-the-shelf solver as proposed in MPM. We hope that the case study reported in this article will favor in the future the integration of research tools in large open source software.

[17] `https://github.com/rpm-software-management/dnf`

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] Mancoosi International Solver Competition website `http://www.mancoosi.org/misc/`

[4] Pietro Abate, Roberto Di Cosmo. Adoption of Academic Tools in Open Source Communities: The Debian Case Study. In *Proceedings of Open Source Systems: Towards Robust Practices - 13th IFIP WG 2.13 International Conference (OSS 2017)*, pp 139–150, 2017.

[5] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, Stefano Zacchiroli. A modular package manager architecture. Information & Software Technology 55(2): 459-474, 2013.

[6] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, Stefano Zacchiroli. Dependency solving: A separate concern in component evolution management. Journal of Systems and Software 85(10): 2228-2240, 2012.

[7] Pietro Abate, Jaap Boender, Roberto Di Cosmo, and Stefano Zacchiroli. Strong dependencies between software components. Technical Report 2, Mancoosi - Seventh Framework Programme, May 2009.

[8] 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.

[9] Josep Argelich, Ines Lynce, and Joao Marques-Silva. On solving Boolean multilevel optimization problems. In *Twenty-First International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 393–398, Pasadena, California, USA, 2009.

[10] 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.

[11] Anton Belov, Inês Lynce, João Marques-Silva. Towards efficient MUS extraction. AI Commun. 25(2): 97-116, 2012

[12] Armin Biere, Edmund M. Clarke, Richard Raimi, Yunshan Zhu. Verifiying Safety Properties of a Power PC Microprocessor Using Symbolic Model Checking without BDDs. In *Proc. of CAV 1999*, pp 60-71, 1999.

[13] Handbook of Satisfiability Armin Biere and Marijn Heule and Hans van Maaren and Toby Walsh Editors Frontiers in Artificial Intelligence and Applications, vol. 185, 2009

[14] 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.

[15] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third IEEE Symposium on the Foundations of Computer Science*, pp 151-158, 1971

[16] Niklas Eén and 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.

[17] Alan M. Frisch and Paul A. Giannoros. SAT encodings of the at-most-k constraint: Some old, some new, some fast, some slow. In *Proceedings of the The Ninth International Workshop on Constraint Modelling and Reformulation (ModRef 2010)*, 2010

[18] Gebser, M., Kaminski, R., Schaub, T. aspcud: a Linux package configuration tool based on answer set programming. In Drescher, C., Lynce, I., Treinen, R., (eds.) *Proceedings Logics for Component Configuration LoCoCo (2011)*, 2011.

[19] Éric Grégoire, Jean-Marie Lagniez, Bertrand Mazure. Boosting MUC extraction in unsatisfiable constraint networks. Applied Intelligence 41(4): 1012-1023, 2014

[20] Éric Grégoire, Bertrand Mazure, Cédric Piette: Using local search to find MSSes and MUSes. European Journal of Operational Research 199(3): 640-646, 2009.

[21] Steffen Hölldobler and Van Hau Nguyen. On SAT-Encodings of the At-Most-One Constraint. In *Proceedings of The Twelfth International Workshop on Constraint Modelling and Reformulation (ModRef 2013)*, Uppsala, Sweden, September 16-20. pp. 1–17, 2013

[22] Daniel Le Berre and Anne Parrain. The Sat4j library 2.2, System Description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[23] Daniel Le Berre and Pascal Rapicault. Dependency Management for the Eclipse Ecosystem. Proceedings of IWOCE2009 - Open Component Ecosystems International Workshop, August 2009.

[24] Chu Min Li and Felip Manyà. MaxSAT, Hard and Soft Constraints. In [13], pp 613-631, 2009

[25] Ines Lynce and Joao P. Marques Silva. On computing minimum unsatisfiable cores. In *SAT*, 2004.

[26] 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.

[27] Olivier Roussel and Vasco M. Manquinho. Pseudo-Boolean and Cardinality Constraints In [13], pp. 695-733, 2009

[28] Claude Michel and Michel Rueher. Handling software upgradeability problems with MILP solvers. In: *Proceedings of LoCoCo2010 - Workshop on Logics for Component Configuration*, 2010.

[29] 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.

[30] Alexander Nadel, Vadim Ryvchin, Ofer Strichman. Accelerated Deletion-based Extraction of Minimal Unsatisfiable Cores. JSAT 9: 27-51, 2014.

[31] Mark Powell (NASA) Marc Hoffmann, Gilles J. Iachelini (CSC). Eclipse on rails and rockets. http://live.eclipse.org/node/750.

[32] Tommi Syrjänen. A rule-based formal model for software configuration. Master's thesis, Helsinki University of Technology, 1999.

[33] Ralph Treinen and Stefano Zacchiroli. Solving package dependencies: from Edos to Mancoosi. In *DebConf'8*, Argentine, 2008.

[34] Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. Opium: Optimal package install/uninstall manager. In *ICSE*, pages 178–188. IEEE Computer Society, 2007.