Journal on Satisfiability, Boolean Modeling and Computation 8 (2012) 89-94

PackUp: Tools for Package Upgradability Solving*

SYSTEM DESCRIPTION

Mikoláš Janota mikolas@sat.inesc-id.pt
Inês Lynce ines@sat.inesc-id.pt
Vasco Manquinho vmm@sat.inesc-id.pt
INESC-ID/IST, Technical University of Lisbon, Portugal
Rua Alves Redol n. 9
1000-029 Lisboa, Portugal

Joao Marques-Silva jpms@ucd.ie
Complex and Adaptive Systems Laboratory (CASL)
University College Dublin, Belfield
Dublin 4, Ireland

Abstract

This paper presents PackUp[1.] (PACKage UPgradability with Boolean formulations) a framework for solving the *the software package upgradability problem*. Earlier versions of the framework (cudf2msu, cudf2pbo) participated in the 3rd MISC-live, an international competition organized by the European project MANCOOSI. The framework encodes the problem as a *weighted partial MaxSAT formula* and invokes a dedicated solver to solve the formula. The framework supports two types of solvers: weighted partial MaxSAT solvers and optimization pseudo-Boolean (OPB) solvers. The paper discusses the design of the framework and the specifics of the problem encoding.

KEYWORDS: *package upgradability problem, MaxSAT, Boolean optimization*

Submitted April 2011; revised June 2011; published January 2012

1. Introduction

Package management systems gained popularity in the last few decades due to the success of Linux distributions by facilitating management of software on an operating system. Indeed, typically a single command is sufficient to install or upgrade a piece of software—a package.

Package management is computationally difficult because of interactions between packages: a package may *depend* on other packages or it might *conflict* with other packages. A *package manager* must maintain the packages in a configuration that satisfies dependencies and does not cause conflicts. Finding such configuration is called the *package upgradability problem* and is known to be NP-complete [5].

Satisfying the requirements of dependencies and conflicts is typically insufficient since users have *preferences* over package configurations. For instance, a user may want to change

* This work is partially supported by SFI PI grant BEACON (09/IN.1/I2618), EC FP7 project MANCOOSI (214898), FCT grants ATTEST (CMU-PT/ELE/0009/2009), BSOLO (PTDC/EIA/76572/2006), iExplain (PTDC/EIA-CCO/102077/2008) and INESC-ID multiannual funding from the PIDDAC program funds.

1. http://sat.inesc-id.pt/~mikolas/sw/packup

the system as little as possible. This motivates package managers that not only maintain correct configurations but also yield configurations optimal with respect to a given criterion.

This paper presents `PackUp`, a framework for solving the package upgradability problem specified in the Common Upgradability Description Format (CUDF) [8]. The underlying technology are Maximum Satisfiability (MaxSAT) and Pseudo-Boolean Optimization (PBO) solving. Two instances of an earlier version of the framework (`cudf2msu`, `cudf2pbo`) participated in the 3rd live MANCOOSI International Solver Competition (MISC Live 3)[2.] where they have jointly won 4 out of 5 tracks.

2. Problem Statement

The package upgradability problem has two parts. One part comprises the *package universe*, which defines a set of packages, their versions, dependencies, conflicts, as well as other information. The second part of the problem is the *user request*, which specifies packages to be installed, removed, updated, etc. The solution to the problem is a subset of the package universe whose installation satisfies constraints between the packages and the user request.

In practice, different languages for describing package upgradability problems have been introduced (e.g. `rpm, debian`). To facilitate evaluation of solvers, the MANCOOSI project developed a standardized format called CUDF (Common Upgradability Description Format); `PackUp` supports CUDF 2.0 [8], the most up-to-date version at the time of writing.

Since the full description of CUDF 2.0 is beyond the scope of this paper, only the prominent features of the format are considered. Each package has a name, version, dependencies, conflicts, recommended packages, and information whether the package is installed or not. A package universe is modeled as a *package description*, which is a partial function from name-version pairs to a tuple of the package's properties. For a package description ϕ, name p, and version v, we write $\phi(p, v).\mathit{installed}$, $\phi(p, v).\mathit{conflicts}$, $\phi(p, v).\mathit{depends}$, and $\phi(p, v).\mathit{recommends}$, for the respective properties of package p with version v.

The *installed* property of a package determines if the package is installed and has either the value *true* or *false*. The other properties hinge on the concept of *constraints*, which are triples (p, relop, n), where p is a package name, n a version number, and *relop* is one of the binary operators $=, \neq, \geq, \leq$. A package description ϕ *satisfies* a constraint (p, relop, n) iff there is a package in ϕ that is installed, has the name p, and version v satisfying $v\,\mathit{relop}\,n$. For instance, $(x, =, 4)$ is satisfied by descriptions where $\phi(x, 4).\mathit{installed} = \mathit{true}$ and $(x, \geq, 4)$ is satisfied by descriptions where $\phi(x, v).\mathit{installed} = \mathit{true}$ for some $v \geq 4$.

The *conflicts* property is a set of constraints corresponding to packages that must not be installed along with the pertaining package, i.e. if $\phi(x, v).\mathit{installed} = \mathit{true}$ then none of the $\phi(x, v).\mathit{conflicts}$ can be satisfied. For instance, $\phi(p, 1).\mathit{conflicts} = \{(x, =, 2), (y, \neq, 3)\}$ means that version 1 of package p conflicts with version 2 of package x and with all the versions of the package y except for version 3. For simplicity, we assume that only $(y, \neq, v)$ is allowed to be a member of $\phi(p, v).\mathit{conflicts}$ when $p = y$.

The *depends* property is a conjunction of disjunctions of constraints under the standard semantics of conjunction and disjunction. Hence, if $\phi(x, v).\mathit{installed} = \mathit{true}$ then $\phi(x, v).\mathit{depends}$ must be satisfied. For instance, $\phi(p, 2).\mathit{depends} = ((x, \geq, 3) \wedge (y, \geq, 3)) \vee$

2. http://www.mancoosi.org/misc-live/20101126/

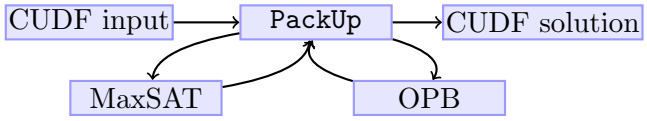

Figure 1. Workflows in PackUp

$(z, \geq, 10)$ means that version 2 of package p requires a version 10, or higher, of package z, or, packages x and y in version at least 3.

The *recommends* property has the same format as *depends* but is not enforced and is used only for expressing optimization criteria.

A *request* is a pair of sets of constraints (l_i, l_d) where l_i determines the packages that must be installed and l_d determines the packages that must be removed. Given a package description ϕ and a request (l_i, l_d), a *solution* to the package upgradability problem is a package description ψ s. t. ψ differs from ϕ only on the *installed* properties; all the *depends* properties are satisfied; no *conflicts* properties are violated in ψ; all constraints in l_i are satisfied and no constraints in l_d are satisfied by the installed packages.

The following text utilizes the following auxiliary definitions. We write $i_\phi(p)$ for the set of versions of a given package, i.e. $i_\phi(p) = \{v \mid (p, v) \in \mathit{Dom}(\phi) \wedge \phi(p, v).\, \mathit{installed} = \mathit{true}\}$. Several measures determine how much a solution ψ changes the original package universe ϕ: number of packages removed, i.e. $\mathit{removed}(\phi, \psi) = |\{p \mid i_\phi(p) \neq \emptyset \wedge i_\psi(p) = \emptyset\}|$, number of new packages, i.e. $\mathit{new}(\phi, \psi) = |\{p \mid i_\phi(p) = \emptyset \wedge i_\psi(p) \neq \emptyset\}|$, number of packages changed, i.e. $\mathit{changed}(\phi, \psi) = |\{p \mid i_\phi(p) \neq i_\psi(p)\}|$, packages not up-to-date, i.e. $\mathit{notuptodate}(\phi, \psi) = |\{p \mid i_\psi(p) \neq \emptyset \wedge v_{\max} \notin i_\psi(p)\}|$, and unsatisfied recommends, i.e. $\mathit{unmet\text{-}recommends}(\phi, \psi)$ is the number of unsatisfied disjunctions in *recommends* property of installed packages in ψ.

A *criterion* is a tuple $(f_1, \ldots, f_n)$ where f_i is one of *removed*, *new*, *changed*, and *notuptodate*, e.g. $(\mathit{removed}, \mathit{new})$. A *score* of a solution ψ for an initial installation ϕ is the tuple $(f_1(\phi, \psi), \ldots, f_n(\phi, \psi))$. Given a package description ϕ, request (l_i, l_d), and a criterion $\mathcal{T}$, a solution is *optimal* iff its score is minimal among all the other solutions w.r.t. lexicographic ordering. For instance, for the criterion $(\mathit{removed}, \mathit{changed})$ a solution ψ_1 is better than ψ_2 iff $\mathit{removed}(\phi, \psi_1) < \mathit{removed}(\phi, \psi_1) \vee (\mathit{removed}(\phi, \psi_1) = \mathit{removed}(\phi, \psi_1) \wedge \mathit{changed}(\phi, \psi_1) < \mathit{changed}(\phi, \psi_1))$.

3. The Framework

Figure 1 schematically depicts the possible workflows in PackUp. The input given in CUDF is encoded into a weighted partial MaxSAT formula. This formula is either solved by a MaxSAT solver or by an OPB solver (which may be called multiple times, see Section 3.2). If the formula is solved, PackUp produces a CUDF solution from the formula's solution.

3.1 Encoding

The encoding is performed in the following sequence of steps.

1. *read in the problem*: reads the problem into dedicated data structures;

Table 1. Definition of the operators $\mathcal{C}$ and $\mathcal{D}$ for constraints.

relop	$\mathcal{C}\,[x, (q, \mathit{relop}, n)]$	$\mathcal{D}\,[x, (q, \mathit{relop}, n)]$
$=$	$\neg x \vee \neg x_q^n$	$\neg x \vee x_q^n$
$\neq$	$(\neg x \vee \mathrm{u}\!\downarrow_q^{n-1}) \wedge (\neg x \vee \mathrm{u}\!\uparrow_q^{n+1})$	$\neg x \vee \mathrm{i}\!\downarrow_q^{n-1} \vee \mathrm{i}\!\uparrow_q^{n+1}$
$\geq$	$\neg x \vee \mathrm{u}\!\uparrow_q^n$	$\neg x \vee \mathrm{i}\!\uparrow_q^n$
$\leq$	$\neg x \vee \mathrm{u}\!\downarrow_q^n$	$\neg x \vee \mathrm{i}\!\downarrow_q^n$

2. *slice*: traverses the data structures obtained in the previous step and discards all packages that are certainly unnecessary to provide a solution [9];
3. *encode package constraints and request*: captures conflicts, depends, and the request;
4. *encode preference*: captures the given preferences;
5. *encode auxiliary variables*: generates additional formulas giving semantics to auxiliary variables used in the previous steps.

The following relies on standard notions from propositional logic, namely *clause* is a disjunction of *literals* and a literal either a Boolean variable or its negation. The problem is encoded as a *weighted partial MaxSAT formula* [4], which comprises two sets of clauses: *hard clauses* and *soft clauses* where each soft clause has a non-negative weight. A solution to such a formula is a variable valuation that satisfies all the hard clauses and maximizes the sum of weights of satisfied soft clauses. A soft clause c with weight W is denoted as (W, c).

Whether a package p with version v is installed or not, is modeled by a Boolean variable x_p^v. Constraints are encoded with the use of the following four types of variables, called *interval variables* (similar to *order encoding* [7]):

- $\mathrm{u}\!\uparrow_p^v$ — all versions greater than or equal to v of p are uninstalled
- $\mathrm{u}\!\downarrow_p^v$ — all versions less than or equal to v of p are uninstalled
- $\mathrm{i}\!\uparrow_p^v$ — at least one version greater than or equal to v of p is installed
- $\mathrm{i}\!\downarrow_p^v$ — at least one version less than or equal to v of p is installed

To define the encoding, we use two auxiliary operators $\mathcal{C}$ and $\mathcal{D}$. The operators correspond to the encoding of conflicts and dependencies, respectively. Table 1 defines the operators at the level of single constraints. So for instance $\mathcal{C}\left[x_p^v, (q, \leq, n)\right]$ yields the formula $\neg x_p^v \vee \mathrm{u}\!\downarrow_q^n$, which represents that if version v of p is installed, then all the packages q with versions less or equal to n must be uninstalled. Analogously, $\mathcal{D}\left[x_p^v, (q, \leq, n)\right]$ yields $\neg x_p^v \vee \mathrm{i}\!\downarrow_q^n$ which represents that if version v of p is installed, then at least one package q with a version less or equal to n must be installed. Observe that $\mathcal{C}\,[x, (q, \mathit{relop}, n)]$ always yields one or two clauses and that $\mathcal{D}\,[x, (q, \mathit{relop}, n)]$ always yields one clause.

To extend $\mathcal{C}$ to a set of constraints l, we take the conjunction of encodings of the constraints in the set. To extend $\mathcal{D}$ to a conjunctive normal form of constraints, we reconstruct the conjunctive normal form from the translations of those constraints:

$$\mathcal{C}[x, l] = \bigwedge\nolimits_{r \in l} \; \mathcal{C}[x, r] \qquad \mathcal{D}[x, l_1 \oplus l_2] = \mathcal{D}[x, l_1] \;\oplus\; \mathcal{D}[x, l_2], \text{where } \oplus \in \{\vee, \wedge\}$$

To give the interval variables their intended meaning, we generate the following clauses for each package p and version v.

$$I_p^v \;=\; (\neg \mathrm{i}\Uparrow_p^v \vee x_p^v \vee \mathrm{i}\Uparrow_p^{v+1}) \;\wedge\; (\neg \mathrm{u}\Uparrow_p^v \vee \neg x_p^v) \;\wedge\; (\neg \mathrm{u}\Uparrow_p^v \vee \mathrm{u}\Uparrow_p^{v+1}) \\ \wedge\; (\neg \mathrm{i}\Downarrow_p^v \vee x_p^v \vee \mathrm{i}\Downarrow_p^{v-1}) \;\wedge\; (\neg \mathrm{u}\Downarrow_p^v \vee \neg x_p^v) \;\wedge\; (\neg \mathrm{u}\Downarrow_p^v \vee \mathrm{u}\Downarrow_p^{v-1})$$

where x_p^v for nonexistent packages is treated as *false*, unneeded interval variables are not generated, and the formulas are simplified accordingly.

To encode the non-preferential part of the upgradability problem comprising a package description ϕ and a request (l_i, l_d), we generate the following formula:

$$r \wedge \mathcal{D}(r, l_i) \wedge \mathcal{C}(r, l_d) \wedge \bigwedge I_p^v \wedge \\ \bigwedge\nolimits_{(p,v)\in Dom(\phi)} \mathcal{D}[x_p^v, \phi(p,v).\, \mathit{depends}] \wedge \mathcal{C}[x_p^v, \phi(p,v).\, \mathit{conflicts}]$$

where r is a fresh variable corresponding to an always-installed package.

To encode preferences, we generate soft clauses capturing the objective to minimize the functions in the given criterion. The weights for these clauses are generated in such a way that they will capture the lexicographic ordering on the scores [1], i.e. for a criterion $\mathcal{T} = (f_1, \ldots, f_n)$ the weight for clauses capturing minimization of the function f_i is defined as $W_i = 1 + \Sigma_{j<i} W_j \times c_j$ where c_j is the number of clauses generated for the function f_j. Hence, in the following we assume that for the functions *removed*, *new*, *changed* and *notuptodate* their corresponding weights W_r, W_n, W_c, and W_u, respectively, were generated for the given criterion $\mathcal{T}$ with $W_i = 0$ if f_i does not appear in $\mathcal{T}$.

Given a package description capturing the initial installation ϕ for the individual functions we use the following rules. For the function *removed*: If $i_\phi(p) \neq \emptyset$ then generate the soft clause $(W_r, \mathrm{i}\Uparrow_p^1)$. For the function *new*: If $i_\phi(p) = \emptyset$ then generate the soft clause $(W_n, \mathrm{u}\Uparrow_p^1)$. For the function *changed*: Let s_p be a fresh variable then generate the following hard clauses $\neg s_p \vee x_p^v$ if $\phi(p,v).\, \mathit{installed} = \mathit{true}$ and $\neg s_p \vee \neg x_p^v$ if $\phi(p,v).\, \mathit{installed} = \mathit{false}$; add the soft clause (W_c, s_p). For the function *notuptodate*: Let t_p be a fresh variable; generate the hard clauses $\neg x_p^v \vee t^p$ for all $(p,v) \in \mathit{Dom}(\psi)$; generate the soft clause $(W_n, \neg t_p \vee x_p^{v_{\max}})$ where $v_{\max}$ is the maximal version of p appearing in ϕ. For the function *unmet-recommends*: For each clause in $c \in \mathcal{D}(x_p^v, \phi(p,v).\, \mathit{recommends})$ generate the soft clause (W_u, c).

3.2 Computing a Solution

Once the problem is encoded as a weighted partial MaxSAT formula, an out-of-the-box solver can be used to solve it. Then, an optimal solution to the upgradability problem is a solution that installs those packages whose corresponding variables have the value *true* in the solution to the formula.

Previous research showed that out-of-the-box solvers do not cope well with large weights resulting from lexicographic ordering [1]. Hence, `PackUp` enables solving the formula iteratively, where each component of the lexicographic ordering is minimized separately (cf [1]). However, this iterative approach requires an OPB solver and a MaxSAT solver needs to implement it internally; this is indeed the case for the solvers `msuncore` and `bmo-pblex`.

We should note that the suitability of the underlying solver may depend on the given criterion. For instance, the solver `msuncore` searches on the *lower-bound* of the optimization

function [3, 6] and therefore is suitable for problems where the optimum is not too far from the best theoretical result. In contrast, `bmo-pblex` and `minisat+` search on the *upper-bound* [2] and therefore are expected to perform well on problems with high deviation from the best theoretical result.

4. Summary

This paper presents the framework `PackUp` for solving the upgradability problem. The core functionality of the system is the encoding of the problem as a weighted partial MaxSAT formula. A somewhat unique feature of this encoding is the use of *interval variables*, which are similar to the *order encoding* used in SAT-based constraint solving [7].

The framework is engineered in such a way that it can be connected to any MaxSAT or OPB solver. As such, together with `minisat+`, `PackUp` provides an open source and free software solution to the package upgradability problem. Last but not least, the architecture enables other researchers to freely experiment with their solvers.

References

[1] Josep Argelich, Daniel Le Berre, Inês Lynce, João P. Marques-Silva, and Pascal Rapicault. Solving Linux upgradeability problems using Boolean optimization. In Inês Lynce and Ralf Treinen, editors, *LoCoCo*, pages 11–22, 2010.

[2] Brian Borchers and Judith Furman. A two-phase exact algorithm for MAX-SAT and Weighted MAX-SAT problems. *J. Comb. Optim.*, **2**(4):299–306, 1998.

[3] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In Armin Biere and Carla P. Gomes, editors, *SAT*, pages 252–265. Springer, 2006.

[4] Chu Min Li and Felip Manyà. MaxSAT, hard and soft constraints. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, pages 613–631. IOS Press, 2009.

[5] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the complexity of large free and open source package-based software distributions. In *ASE*, pages 199–208, 2006.

[6] Vasco M. Manquinho, João P. Marques-Silva, and Jordi Planes. Algorithms for weighted Boolean optimization. In Oliver Kullmann, editor, *SAT*, pages 495–508. Springer, 2009.

[7] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints*, **14**(2):254–272, 2009.

[8] Ralf Treinen and Stefano Zacchiroli. Common upgradeability description format (CUDF) 2.0. Technical Report 003, MANCOOSI, November 2009.

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