mancoosi
managing software complexity

Final version of the optimizations algorithms and tools
Deliverable 4.3

Nature : Deliverable
Due date : 23.05.2011
Start date of project : 01.02.2008
Duration : 40 months

SEVENTH FRAMEWORK PROGRAMME

Information Society Technologies

Specific Targeted Research Project
Contract no.214898
Seventh Framework Programme: FP7-ICT-2007-1

A list of the authors and reviewers

Project acronym	Mancoosi
Project full title	Managing the Complexity of the Open Source Infrastructure
Project number	214898
Authors list	Gustavo Guttierez <gutierrez.gustavo@uclouvain.be> Mikoláš Janota <mikolas@sat.inesc-id.pt> Inês Lynce <ines@sat.inesc-id.pt> Olivier Lhomme <Olivier.Lhomme@fr.ibm.com> Vasco Manquinho <vmm@sat.inesc-id.pt> Joao Marques-Silva <jpms@ucd.ie> Claude Michel <Claude.Michel@i3s.unice.fr>
Workpackage number	WP4
Deliverable number	3
Document type	Deliverable
Version	1
Due date	23/05/2011
Actual submission date	23/05/2011
Distribution	Public
Project coordinator	Roberto Di Cosmo

Abstract

The Mancoosi project aims at solving the upgradeability problem that users of Free and Open Source Software distributions experience when trying to install, remove, or upgrade packages. The specific aim of Workpackage 4 was to study and develop specialized upgradeability solvers.

A first insight of the algorithms and tools to solve the upgradeability problem has be given in deliverable D4.2 [ALLM10]. This new document enhances this description with the new developments that have occurred during the last period of the Mancoosi project.

Contents

1 Introduction **9**

2 Solving the upgradeability problem **11**

2.1 Upgradeability problem as multi-criteria optimization . . . 11

2.1.1 Multicriteria methods without tradeoff . . . 12

2.1.2 Multicriteria methods with tradeoff . . . 12

Weighted sum . . . 12

Maximizing the minimun . . . 13

Dynamic ordering of the criteria . . . 13

Ordered weighted average . . . 13

Leximin/leximax . . . 13

2.2 Algorithms for solving the upgradeability problem . . . 14

2.2.1 Implementation of Lexicographic optimization . . . 14

2.2.2 Implementation of Leximin/Leximax optimization . . . 15

3 Handling multiple criteria in an integer programming framework **17**

3.1 Introduction . . . 17

3.2 Criteria . . . 17

3.2.1 Removed criteria . . . 17

3.2.2 Changed criteria . . . 18

3.2.3 Notuptodate criteria . . . 18

3.2.4 Nunsat criteria . . . 19

3.2.5 New criteria . . . 19

3.2.6 Count criteria . . . 20

3.3 Combining multiple criteria . . . 20

3.3.1 Basic combination . . . 20

Agregation . . . 20

Lexicographic order . . . 20
Leximin/leximax order . . . 21
3.3.2 Extensive combination . . . 21
3.3.3 A small language for user defined combination . . . 22
3.4 Solver modifications and extension . . . 24
3.4.1 Solver implementation modification . . . 24
3.4.2 Criteria and combiner implementation . . . 24

4 PackUP **27**
4.1 Preliminaries . . . 27
4.2 Encoding . . . 29
4.3 Computing a Solution . . . 31

5 Constraint programming and the package installability problem **33**
5.1 The solver infrastructure . . . 33
5.1.1 KCUDF . . . 33
5.1.2 Reduction process . . . 36
5.2 Constraint based solver for the PIP using binary relations . . . 37
5.2.1 Constraint model . . . 37
Variables . . . 37
Constraints . . . 38
5.2.2 Traversal of the search space . . . 39
5.2.3 Redundant constraints . . . 40
Heuristics . . . 41
5.2.4 Optimization . . . 41
5.2.5 Current results . . . 42

List of Figures

2.1 lexicographic optimization 14

2.2 leximin optimization 16

4.1 Workflows in `PackUp` 28

5.1 Solver infrastructure and supporting components. 35

5.2 Reducer state diagram. 36

Chapter 1

Introduction

Mancoosi workpackage 4 purpose was to develop tools and algorithmsin order to solve the upgradeability problem. In other words, from an upgradeability problem described using the CUDF format [TZ09a], WP4 tools compute the best solution according to a set of multicriteria defining user preferences. This problem is harder than the installability problem which has proven to be NP-hard [MBC+06]. Thus, one of the main WP4 issue was to find the right techniques to solve the upgradeability problem in a reasonable amount of time.

WP4 efforts have resulted in the development of four CUDF solvers:

- apt-pbo from CAIXA magica,
- PackUP from INESC-ID,
- cpp-kcudf from UCL,
- and mccs from UNS.

All these solvers have explored different techniques ranging from pseudo boolean solvers to integer programming techniques with various success. Indeed, CUDF problems offer different structures which are more or less fitted to the underlying solver technique.

Note that apt-pbo is not described in the report. This solver is a pseudo boolean solver which have benefited from the work and the know how of INESC-ID. As a result, it uses techniques similar to the CUDF solver developed at INESC-ID. However, apt-pbo has been included in deliverable D4.3 as it represents a clear effort to integrate CUDF solvers in a linux distribution.

The report is organized as follows:

- chapter 2 gives a detailed picture of the study of multicriteria issues done at IBM.
- chapter 3 focuses on the implementation of multicriteria combination within the mccs solver from UNS, an integer programming based framework to solve the upgradeability problem.
- chapter 4 describes the last developments of the PackUP solver from INESC-ID, a framework to solve the upgradeability problem relying on weighted partial MaxSAT solvers.
- chapter 5 introduces the ccp-kcudf solver, a constraint based solver which relies on new techniques to handle binary relations that are the origins of the CUDF problem.

Chapter 2

Solving the upgradeability problem

2.1 Upgradeability problem as multi-criteria optimization

The upgradeability problem can be expressed as a mathematical model as follows:

- A boolean variable is associated to each versioned package.
- Dependencies or incompatibilities between packages or versioned packages are constraints on the corresponding boolean variables.
- Requirements, like *install package X* are also modelized as constraints.
- Measures of the quality of a solution to the upgradeability problem are modelized as objective functions.

In general, we have several measures of the quality of the solutions of the upgradeability problem. For example, we may want to minimize the download time, but in the meantime we want to get the most recent versions of packages and also to minimize the perturbation to the current configuration. Solving the upgradeability problem amounts to solve a combinatorial optimization problem: find values for the variables that maximize the criteria and satisfy the constraints. However, these different measures are often contradictory: clearly, on an old intallation, we can not simultaneously minimize the download time and maximize the recency of the versions of the packages. That is to say that the upgradeability problem, in its generality, is a multi-criteria optimization problem.

The problem is not only to find a good or an optimal solution. Indeed, before solving the problem, we first need to define what kind of solution we want. If we take again the above example of an old intallation, for which we consider two criteria: minimize the download time, and maximize the recency of the versions of the packages, we need to decide what kind of tradeoff between the two criteria we want to have. For example, a solution with an acceptable tradeoff may be a download time of 5 minutes, and the choice to update some specific more important packages first. In general, the ability to express the kind of tradeoff we want depends on the method chosen for solving the multi-criteria optimization problem.

2.1.1 Multicriteria methods without tradeoff

The simpliest approach, the *lexicographic optimality*, defines a first kind of tradeoff: do not make any tradeoff. It amounts to prefer any improvement of the most important criterion, even a very small improvement, to any improvement of the second criterion, even a huge improvement. We have a strong hierarchical preference between criteria. In other words, the criteria are ordered lexicographically: a solution s_1 is better than s_2 iff it is equivalent for criteria 1..i-1, and is better for the i-th criterion. This approach has a great advantage, it does not need the values of the criteria to be compared. Indeed, the comparison of the values of different criteria is a difficult task, and can be specific to a user, and a problem.

Another approach, avoiding the difficult task of defining how the values of different criteria can be compared, is simply to generate all the solutions which may be prefered. A solution may be prefered if and only if it is not dominated by another one. Such a non dominated solution is called a *pareto-solution*. A solution s_1 is better than s_2 iff for each criterion, s_1 is at least equivalent to s_2, and there exists a criterion such as s_1 is strictly better than s_2. Intuitively, we cannot improve a criterion of a pareto-solution without a loss for another criterion. In practice, the set of pareto solutions is huge, and the usual approach is to compute an approximation of this set.

2.1.2 Multicriteria methods with tradeoff

The comparison of the values of different criteria is needed in all multi-criteria methods in order to define what a good tradeoff is. The lexicographic approach, indeed, defines the tradeoff as no possible tradeoff, and therefore does not need such a comparison. In general, this comparison is made thanks to a *utility function* associated to each criterion. A utility function u_z maps a value v of the criteria z to a number $u_z(v)$. A utility function u_z should satisfy an intuitive property: $v_1 < v_2 \rightarrow u_z(v1) < u_z(v2)$.

The aim of utility functions is to make possible the comparison between different criteria. The utility functions of different criteria are strongly related, in such a way that the numbers coming from different criteria can be compared: let z_1, z_2 be two criteria and v_1, v_2 be the values of z_1, z_2 in a given solution. We can know that, in this solution, the criterion z_1 is more favoured than the criterion z_2 when $u_{z_1}(v_1) > u_{z_2}(v_2)$.

Given the values $u_1, \ldots, u_k$ and $u'_1, \ldots, u'_k$ of the utility functions for two solutions s and s', we want now to define when s is prefered to s', noted $s >_m s'$. Many different preference orders $>_m$, and many corresponding multi-criteria methods exist. For the upgradeability problem, we considered the following as particularly interesting methods.

Weighted sum

An extremely frequent approach is to maximize a weighted sum $\sum w_i * u_i$, where $w_1, \ldots, w_k$ are weights associated to each criterion.

Formally, we have: $s >_m s'$ *iff* $\sum w_i * u_i > \sum w_i * u'_i$.

This approach seems to be quite intuitive and satisfactory. Nevertheless, despite its general use in many domains, this approach has some strong drawbacks. The function $\sum w_i * u_i$ is linear, and, when maximizing this function for a concave set of solutions, many good solutions are not reachable. Furthermore, its sensitivity to the weights is very high.

Maximizing the minimun

A different approach, which has many applications too, is to maximize the minimum value among the criteria. $s >_m s'$ *iff* $\min u_i > \min u'_i$.

This approach unfortunately produces many undistinguishable solutions, and some of them are not pareto solutions.

Dynamic ordering of the criteria

The weighted sum method, as well as the method of maximizing the minimun have clear drawbacks. Indeed, these two quite different methods can be strongly improved by integrating in them the same following idea. This idea is to order the criteria depending on the values they take in a solution. Then, aggregation of the criteria can be done depending on the rank of the criteria. The weighted sum method, where the weights depend on the rank of the criteria in the dynamic ordering, becomes the *ordered weighted average* method. The method of maximizing the minimun can be extended in the case of a tie: when the minimum in the two solutions to compare are equal, the extended method takes into account the second minimum, and if necessary the third minimum, the fourth one, and so on. This extension gives the *leximin* approach. These two methods are decribed below.

Ordered weighted average

Let $p_1, ..., p_k$ be the sorted permutation of $u_1, \ldots, u_k$ in increasing order of values. Let w be a vector of weights such that $w_i \geq w_{i+1}$.

The idea is to maximize $\sum w_i * p_i$. We can see that if $w_1 = w_2 = \ldots = w_k$, we get the sum. If $w_1 = 1, w_2 = w_3 = \ldots = 0$, we get the methods which maximize the minimum.

This method has been introduced by [Yag88].

Leximin/leximax

Once again, let $p_1, ..., p_k$ be the sorted permutation of $u_1, \ldots, u_k$ in increasing order of values.

Then a Leximin solution is simply a lexicographic optimal solution on $p_1, ..., p_k$.

Leximin/leximax have nice properties, that make them intuitive for user:

- They produce only pareto solutions;
- They are egalitarist: a leximin solution is always a solution for the method which maximizes the minimum;
- They verify the property of reducibility: if a criteria is set to a value it has in an optimal solution, we get the same set of optimal solutions.

This method has been introduced in social choice community and then used for multicriteria optimization [Ehr00]. An efficient algorithm has been proposed in [Ogr97]. Note also a clear presentation of this method in [BL09], where different algorithmic adaptations for constraint programming are described.

2.2 Algorithms for solving the upgradeability problem

Within the Mancoosi project, we focused the work on a basic multicriteria approach without tradeoff, the lexicographic opimization, and on a method with tradeoff that seems well adapted to upgradeability problems: the leximin approach. The next sections present algorithms for implementing these approaches.

2.2.1 Implementation of Lexicographic optimization

Different possibilities exist for implementating a lexicographic optimization. In this section, we describe a general approach that can be implemented on top of any existing optimization solver, but which does not depend on the underlying solver.

The underlying optimization solver is only required to be able to maximize an objective function F subject to the set of constraints C.

Let $max(F, C)$ be the minimal value of F among all the feasible solutions of the constraint system C. This optimal value is computed by calling the underlying optimization solver.

Given n criteria $F_1, \ldots, F_n$, and a a set of constraints C , the lexicographic optimization problem amounts to maximize lexicographically $F_1, \ldots, F_n$ while satisfying the set of constraints C.

A lexicographic optimization problem P can be solved by solving a sequence of (single criterion) optimization problems $\{P_i\}_{i\in[1,n]}$ with an underlying optimization solver.

Each problem P_i consists of maximizing F_i while satisfying the constraints C_i where:

- $C_1 = C$,
- for $i \in [2, n], C_i = C_{i-1} \cup \{F_{i-1} = max(F_{i-1}, C_{i-1})\}$

The solution of the optimization problem $\{P_n\}$ is the solution of the lexicographic optimization problem P.

We can see that solving the problem P_i needs the optimal value of the problem P_{i-1}. Therefore, the algorithm for computing the solution of the lexicographic optimization problem simply amounts to compute the solutions of the subproblems $\{P_i\}_{i\in[1,n]}$ with the underlying optimization solver in the order P_1, then $P_2, \ldots,$ until P_n (see Figure2.1).

```
procedure Lexicographic Maximize(F_1, ..., F_n, C)
1     for i := 1..n
          //define problem P_i
2         if i>1
3            C:=C ∪ {F_{i-1} = max}
          //solve problem P_i
4         maximize(F_i, C) with the underlying solver
5         max:= max(F_i, C)
6     print solution of problem P_n
```

Figure 2.1: lexicographic optimization

2.2.2 Implementation of Leximin/Leximax optimization

As for lexicographic optimization, leximin/leximax optimization relies on the existence of an optimization solver. The approach described in this section is general and can be implemented on many existing optimization solvers. The approach is presented in a form compatible with a linear solver with 0-1 variables.

The underlying optimization solver is required to be able to maximize an objective function F subject to the set of constraints C, and also to be able to handle sum constraints.

Let $max(F, C)$ be the minimal value of F among all the feasible solutions of C. This optimal value is computed by calling the underlying optimization solver.

A leximin optimization problem P, defined by its n criteria $F_1, \ldots, F_n$, and its set of constraints C, can be solved by solving a sequence of (single criterion) optimization problems $\{P_i\}_{i\in[1,n]}$ with the underlying optimization solver. The subproblems P_i are defined as follows.

Subproblem P_i needs to maximize v_i while satisfying the constraints C_i where:

- v_i is a new variable which is contrained to be the $i - th$ worst criterion by:
  1. Constraining it to be less or equal than at least $n - i + 1$ criteria, and,
  2. Maximizing it.
- $C_1 = C \cup \{v_1 \leq F_1, \ldots, v_1 \leq F_n\}$
- for $i \in [2, n], C_i =$

$$C_{i-1}$$

$$\cup$$

$$\{v_i \leq F_1 + a_{i,1}M, \ldots, v_i \leq F_n + a_{i,n}M\}$$

$$\cup$$

$$\{\sum_{j=1..n} a_{i,j} \leq i - 1\}$$

$$\cup$$

$$\{v_{i-1} = max(v_{i-1}, C_{i-1})\}$$

  where:

  - M is a big number.
  - $a_{i,j}$ are 0-1 variables:
    * $a_{i,j} = 1$ means that the constraint $v_i \leq F_j + a_{i,j}M$ is trivially true, and, thus, that $v_i \leq F_j$ is not enforced.
    * Conversely, $a_{i,j} = 0$ means that the constraint $v_i \leq F_j + a_{i,j}M$ can be rewritten as $v_i \leq F_j$ which, therefore, is enforced.

  Note that the sum constraint, $\sum_{j=1..n} a_{i,j} \leq i-1$, forces at least $n-i+1$ constraints of the form $v_i \leq F_j$ to be satisfied.

The solution of the optimization problem $\{P_n\}$ is the solution of the leximin optimization problem P.

Solving the problem P_i needs the optimal value of the problem P_{i-1}. Therefore, the algorithm for computing the solution of the leximin optimization problem amounts to compute the solutions of the subproblems $\{P_i\}_{i \in [1,n]}$ with the underlying optimization solver in the order P_1, then $P_2, \ldots,$ until P_n (see Figure 2.2).

```
procedure Leximin(F_1, ..., F_n, C)
1       for i := 1..n
            //define problem P_i
2           if i=1
3               C:=C ∪ {v_1 ≤ F_1, ..., v_1 ≤ F_n}
4           else
5               C:=C ∪ {v_i ≤ F_1 + a_{i,1}M, ..., v_i ≤ F_n + a_{i,n}M}
6               C:=C ∪ {∑_{j=1..n} a_{i,j} ≤ i − 1}
7               C:=C ∪ {v_{i−1} = max}
            // solve problem P_i
8           maximize(v_i, C) with the underlying solver
9           max:= max(v_i, C)
10      print solution of problem P_n
```

Figure 2.2: leximin optimization

Note that the line 3 of the algorithm, handling the case of $i = 1$, is a simple rewriting of the general case: it corresponds to the general case where all the $a_{1,j}$ are equal to 0, and thus all the constraints $v_1 \leq F_j$ are to be satisfied.

Chapter 3

Handling multiple criteria in an integer programming framework

3.1 Introduction

mccs, which stands for Multi Criteria CUDF Solver, provides an integer programming platform to solve CUDF problems as defined in the Mancoosi project[1]. It translates a CUDF problem and a set of criteria in one or more integer programs that are then solved by an underlying integer programming solver. **mccs** has been developped at UNS in C++ and offers interfaces to many integer programming, as well as pseudo boolean, solvers.

The main features of this solver[2] have alredy been described in section 4 of [ALLM10]. Therefore, this chapter focuses on a new feature of **mccs**: its capability to let the user combine criteria in different ways. The first solver version was only capable to combine criteria in a lexicographic order. Though such an order is useful, the user might want to combine its criteria in a less strict order that do no give a so strong priority to the first criterion. For this purpose, **mccs** has been enhanced with the capability to handle criteria in a leximin/leximax order, as well as, a more simple agregate of criteria. These criteria combinations are the basis of a particularly fexible system of criteria combination.

3.2 Criteria

As a preliminary, this section describes the criteria available in **mccs**. Among the six available criteria, five of them have been introduced in the Mancoosi competition. The last one is aimed at choosing among the solutions the one which, for example, reduces the installation size or the bandwith required to download the new packages.

3.2.1 Removed criteria

removed counts the number of packages r_p which have been removed from the initial configuration $\mathcal{C}_i$ in the final configuration $\mathcal{C}_f$. Let $\mathcal{I}(\mathcal{C}_i)$ be the set of packages with at least one

[1]See http://www.mancoosi.org/

[2]Note that we changed the solver name from CUDFsolver to **mccs** to avoid confusion with other solvers.

installed version in the initial configuration and $\mathcal{V}(p)$ be the set of p versions, then, $n_{removed}$, the number of removed feature is

$$n_{removed} = \sum_{p \in \mathcal{I}(\mathcal{C}_i)} r_p$$

with each r_p being subject to the two following constraints

$$r_p + \sum_{v \in \mathcal{V}(p)} p^v \geq 1$$

and

$$|\mathcal{V}(p)| r_p + \sum_{v \in \mathcal{V}(p)} p^v \leq |\mathcal{V}(p)|$$

where the first constraint forces r_p to 1 if none of p versions is installed and the second constraints sets r_p to zero if at least one of p versions is installed.

3.2.2 Changed criteria

changed: counts the number of packages which have a modified set of version in the solution with respect to the initial configuration. Let $\mathcal{IV}(p)$ be the set of installed versions of p in $\mathcal{C}_i$ and $\mathcal{UV}(p)$ be the set of uninstalled versions of p in $\mathcal{C}_i$, then $n_{changed}$ is given by

$$n_{changed} = \sum_{p \in \mathcal{C}_i} c_p$$

with each c_p being subject to the two following constraints

$$-c_p - \sum_{v_i \in \mathcal{IV}(p)} p^{v_i} + \sum_{v_u \in \mathcal{UV}(p)} p^{v_u} \geq -|\mathcal{IV}(p)|$$

and

$$-|\mathcal{V}(p)| c_p - \sum_{v_i \in \mathcal{IV}(p)} p^{v_i} + \sum_{v_u \in \mathcal{UV}(p)} p^{v_u} \leq -|\mathcal{IV}(p)|$$

where the sum $-\sum_{v_i \in \mathcal{IV}(p)} p^{v_i} + \sum_{v_u \in \mathcal{UV}(p)} p^{v_u}$ will increase if a version of package p installed in the initial configuration is uninstalled in the final configuration or if a version of package p uninstalled in the initial configuration is installed in the final one. Note that, in the initial configuration, this sum is equal to $-|\mathcal{IV}(p)|$, the cardinality of the set of installed versions of p. As a consequence, the first constraint sets c_p to zero if no change in the installed versions occurs while the second constraint forces c_p to one on any change.

3.2.3 Notuptodate criteria

notuptodate counts the number of installed packages which are not installed in their latest available configuration. Let $sup(\mathcal{V}(p))$ be the highest available version of p, then $n_{notuptodate}$ is given by

$$n_{notuptodate} = \sum_{p \in \mathcal{C}_i} nu_p$$

with each nuc_p being subject to the two following constraints

$$-|\mathcal{V}(p)|nu_p - (|\mathcal{V}(p)|-1)p^{sup(\mathcal{V}(p))} + \sum_{v \in \mathcal{V}(p) - sup(\mathcal{V}(p))} p^v \leq 0$$

and

$$-|\mathcal{V}(p)|nu_p - (|\mathcal{V}(p)|-1)p^{sup(\mathcal{V}(p))} + \sum_{v \in \mathcal{V}(p) - sup(\mathcal{V}(p))} p^v \geq -|\mathcal{V}(p)| + 1$$

where the first constraint sets nu_p to one if one of p version is installed while the highest version of p is not installed, and the second constraint sets it to zero if the highest version of p is installed or none of the lower versions of p are installed.

3.2.4 Nunsat criteria

Nunsat counts the number of disjunctions in a vpkgformula property of installed packages that are not satisfied by final configuration.

$$n_{nunsat} = \sum_{p \in \mathcal{C}_i} \sum_{v \in \mathcal{V}(p)} \sum_{d \in \mathcal{D}isjunct(property(p^v))} ur_d$$

with each ur_d being subject to the following constraints

$$ur_d - p^v \leq 0$$

$$ur_d - ns_d \leq 0$$

$$ur_d - p^v - ns_d \geq -1$$

where ns_d is true iff d is unsatisfied

$$ns_d + \sum_{p_d^v \in d} p_d^v \geq 1$$

$$|d|ns_d + \sum_{p_d^v \in d} p_d^v \leq |d|$$

Here, the two first constraints set ur_d to 0 when p^v, i.e., version v of package p is not installed or the disjunct is satisfied (i.e., $ns_d = 0$). The third constraint sets ur_d to one if p^v is installed and the disjunct is unsatisfied (i.e., $ns_d = 1$). The fourth constraint sets ns_d to 1 (i.e., unsat) if none of package p version is installed while the last constraint sets it to 0 (i.e., sat) if one or more version of p are installed.

3.2.5 New criteria

new counts the number of newly installed packages. Let $\mathcal{U}(\mathcal{C}_i)$ be the set of packages with none installed version in the initial configuration, then, n_{new}, the number of removed feature is given by

$$n_{new} = \sum_{p \in \mathcal{U}(\mathcal{C}_i)} new_p$$

with each new_p being subject to the two following constraints

$$-new_p + \sum_{v \in \mathcal{V}(p)} p^v \geq 0$$

and

$$-|\mathcal{V}(p)|new_p + \sum_{v \in \mathcal{V}(p)} p^v \leq 0$$

where the first constraint sets new_p to 0 if no version of p is installed while the second constraint sets it to 1 if at least one version of p is installed.

3.2.6 Count criteria

count computes the weight of a given *property* on installed versionned packages

$$count = \sum_{p \in \mathcal{C}_i} \sum_{v \in \mathcal{V}(p)} value(property) * p^v$$

Using an option, the set of handled packages can be reduced to the set of newly installed pacakages. In such a case, the previous function will be reduced to the set of uninstalled packages in the initial configuration.

3.3 Combining multiple criteria

All the underlying integer programming solvers used to solve CUDF problems only do monoobjective optimisation. As a consequence, solving multicriteria problems requires a mean to translate the initial multicriteria problem in a set of monocriteria problems. This section describes the different options that **mccs** offer the possiblity to carry out such a translation.

3.3.1 Basic combination

Agregation

The easiest way to combine multiple criteria is through a simple agregation of the criteria. Let c_i be n criteria and λ_i be the user defined weight of criteria c_i, the objective function of a criteria agregation is given by:

$$obj = \sum_{i=1}^{n} \lambda_i * c_i$$

Though easy to define and to solve, a criteria agregation is not always easy to understand. For instance, assuming that criteria c_1 has weight 1 and criteria c_2 has weight 2, decreasing c_1 by 2 is equivalent to decreasing c_2 by 1. This behavior can lead to consider two very different solutions as equivalent.

Lexicographic order

The lexicographic order allows a clear and well defined behavior of the solver: each criteria is ordered according to its lexicographic specification, i.e., the highest priority is given to the first specified criteria . The two common ways to solve a lexicographic order are by using an algorithm implementation or a lexicographic agregate of the criteria.

Algorithm implementation optimises each criteria after the other in order to reach the lexicographic order. The first criteria c_1 is optimised subject to the problem constraint set P, i.e., dependency and conflict constraints:

$$\begin{aligned} & min\ c_1 \\ subject\ to & \\ & P \end{aligned}$$

Then, a new constraint is introduced to insure that the first criteria is equal to the computed objective function value O_1, the objective function is set to c_2 and the problem is solved again to get O_2, the optimal value of c_2. This process is repeated for each criteria:

$$\begin{aligned} & min\ c_i \\ subject\ to & \\ & c_1 = 0_1 \\ & \dots \\ & c_{i-1} = O_{i-1} \\ & P \end{aligned}$$

Thus, this algorithm which requires solving one optimisation problem for each criteria could be costly.

Lexicographic agregate is another way to implement a lexicographic order. Instead of calling the underlying solver for each criteria, they are agregated in a single objective function:

$$obj = \sum_{i=1}^{n} \lambda_i * c_i$$

where

$$\lambda_i > sup(\sum_{j=i+1}^{n} \lambda_j * c_j)$$

The key issue here is to correctly choose criteria weights so that any change in $c_{i+1}, \dots, c_n$ criteria values does not affect c_i. **mccs** computes such weights. However, all solvers have limited domain values for their objective functions. So, in practice, combining more than 2 to 3 criteria in a lexicographic agregate can lead to a solver failure.

Leximin/leximax order

The leximin/leximax order offers more balanced solutions than the lexicographic order. It is suitable when the user does not want to give a strict priority to some criteria. **mccs** implementation of the leximin/leximax has been described in section 2.2.2.

Note that this algorithm relies on integer variables which are not available in pseudo boolean solvers. Thus, the use of leximin/leximax order is restricted to actual integer programming solvers.

3.3.2 Extensive combination

The criteria combination introduced in the previous section offers much more possible criteria combination than expected. A key observation here is that all these combiners can be viewed as

a variation of the lexicographic algorithm: the agregate and lexagregate combiners use only one objective function which could be solved using the lexicographic algorithm. The leximin/leximax combiners are implemented using such an algorithm. These properties have been used to extend criteria combination capabilities.

Both agregate and lexagregate combine their criteria through a single linear combination which only requires one call to the underlying integer programming solver. A lexicographic algorithm might solve one of these combination but, it can also solve more than one of these combination. For example, the trendy set of criteria is composed of four different criteria which have to be solved in a lexicographic order. Solving the four criteria using the lexicographic algorithm requires four calls to the underlying solver, and that is a quite time consumming approach. Solving them using the lexagregate combiner requires a linear combination of the criteria which does not fit to the objective function domain limitations of the underlying solver. A compromise could be reached by combining the two previous combiners, i.e., by using a lexicographic algorithm on two lexagregate pairing the criteria two by two. This approach preserves the combination semantics (it does select the same set of solutions) while improving time consumption.

Combining agregate and lexagregate together or between themselves is also possible and it results in a linear combination of linear combinations. Such combinations can then be used like criteria within a lexicographic or a leximin/leximax order.

Finally, the leximin/leximax combiners offer less flexibility. They can only be combined with a lexicographic order. However,they provide a useful mean to address some user's needs. For example, assuming that three criteria have to be combined and that, while the first criteria is critical, the two next ones are quite equivalent to the user. Then, applying a lexicographic algorithm on the first criteria, followed by a leximin/leximax order on the two last criteria will fit with users needs.

3.3.3 A small language for user defined combination

mccs provides a small dedicated language to let the user define its own criteria combination. The key idea is to map, at least, virtually, any possible criteria combination to an underlying lexicographic order. Indeed, monoobjective criteria combinations, like a simple agregation, fits directly to a lexicographic algorithm reduced to one criteria and the leximin/leximax order implementation do rely on an underlying lexicographic order implementation. Therefore, the basic criteria combination is the lexicographic one:

```
-lexicographic[<lccriteria>{,<lccriteria>}*]
```

The next level of criteria combination gives access to the leximin/leximax order:

```
<lccriteria> ::= {+,-}leximax[<ccriteria>{,<ccriteria>}*] |
                 {+,-}leximin[<ccriteria>{,<ccriteria>}*] |
                 <ccriteria>
```

As a consequence, one, two or more leximin/leximax orders can be lexicographically ordered, meaning that, for instance, the user does not want to order the two first criteria, neither he wants to order the third and fourth criteria, while he clearly wants the set of the two first criteria to be handled before the rest of the criteria.

Note that the leading sign gives the optimisation direction, i.e., a leading `-` for a minimisation and a leading `+` for a maximisation.

The final level of combiners gives access to the agregate and lexagregate combination:

```
<ccriteria> ::= {+,-}agregate[<ccriteria>{,<ccriteria>}*]{[lambda]}? |
                {+,-}lexagregate[<ccriteria>{,<cccriteria>}*]{[lambda]}? |
                <criteria>
```

This recursive definition allows the user to define agregate of agregate or lexagregate. An optional lambda value gives the opportunity to attach a weight to the agregate by means of a positive integer. All the coefficients of the criteria will be multiplied by this weight.

At last, the user can choose among the following criteria which one he wants to optimize:

```
<criteria> ::= {+,-}removed{[lambda]}? |
               {+,-}changed{[lambda]}? |
               {+,-}notuptodate{[lambda]}? |
               {+,-}new{[lambda]}? |
               {+,-}nunsat[<property:>,<withproviders>]{[lambda]}?
               {+,-}count[<property:>]{[lambda]}?
```

With such a language, the trendy set of criteria can be defined using a pure lexicographic order:

```
-lexicographic[-removed,-notuptodate,-nunsat[recommends:,true],-new]
```

or a combination of lexicographic order and lexicographic agregate to balance the number of optimisation to do and the limitation of the solver to handle big values:

```
-lexicographic[-lexagregate[-removed,-notuptodate],
                         -lexagregate[-nunsat[recommends:,true],-new]]
```

Note that, spaces can not be used within a criteria combination.
This final example shows how weights could be used to give more priority to some criteria:

```
-lexicographic[-removed,-agregate[-count[size:,true][10],-count[installedsize:,true]]]
```

To facilitate criteria combination definition by the user, the following criteria combination shortcuts have been introduced:

- -lex[<lccriteria>{,<lccriteria>}*] which is equivalent to
  -lexicographic[<lccriteria>{,<lccriteria>}*]
- -lexagregate[<ccriteria>{,<ccriteria>}*] which is equivalent to
  -lex[-lexagregate[<ccriteria>{,<ccriteria>}*]]
- -lexsemiagregate[<ccriteria>{,<ccriteria>}*] which is equivalent to
  -lex[-lexagregate[<ccriteria>,<ccriteria>],-lexagregate[<ccriteria>,<ccriteria>],...]
- -agregate[<ccriteria>{,<ccriteria>}*] which is equivalent to
  -lex[-agregate[<ccriteria>{,<ccriteria>}*]]
- -leximax[<ccriteria>{,<ccriteria>}*] which is equivalent to
  -lex[-leximax[<ccriteria>{,<ccriteria>}*]]
- -leximin[<ccriteria>{,<ccriteria>}*] which is equivalent to
  -lex[-lexmin[<ccriteria>{,<ccriteria>}*]]
- -lexleximax[<ccriteria>{,<ccriteria>}*] which is equivalent to
  -lex[<ccriteria>,-leximax[<ccriteria>{,<ccriteria>}*]]

- -lexleximin[<ccriteria>{,<ccriteria>}*] which is equivalent to
  -lex[<ccriteria>,-leximin[<ccriteria>{,<ccriteria>}*]]

Thus, for example, instead of writing
-lex[-lexagregate[-removed,-notuptodate],-lexagregate[-nunsat[recommends:,true],-new]]
to describe the trendy set of criteria, the user can more simply write
-lexsemiagregate[-removed,-notuptodate,-nunsat[recommends:,true],-new]
or, instead of
-lex[-removed,-leximax[-count[installedsize:,true],-count[size:,true]]]
he could more simply write
-lexleximax[-removed,-count[installedsize:,true],-count[size:,true]]

3.4 Solver modifications and extension

A first description of the **mccs** architecture was made in [ALLM10]. This section describes the main modifications that have been done to the solver to cope with its new multicriteria capabilities.

3.4.1 Solver implementation modification

The support of a lexicographic algorithm have required changes in the solver interface implementation. The previous interface was relying on a monocriteria approach and thus, had only to deal with a unique objective function. Now that **mccs** allows the handling of multiple criteria and flexible combination of these criteria, a deep modification of the solve function of each solver interface was required. To this regard, the abstract_solver class was modified to allow the handling of a list of criteria and, each solver implementation was modified to handle a lexicographic order in its solve method.

The abstract_solver class is implemented by the following classes:

- cplex_solver.h & cplex_solver.c which implements an interface with the Cplex solver.
- gurobi_solver.h & gurobi_solver.c which implements an interface with the Gurobi solver.
- lpsolve_solver.h & lpsolve_solver.c which implements an interface with the Lpsolve solver.
- glpk_solver.h & glpk_solver.c which implements an interface with the GLPK solver.
- lp_solver.h & lp_solver.c which implements an interface with lp format compliant solvers. Note that the lp format used here is the one defined by cplex and that it has only be tested with the cbc and scip solvers.
- pblib_solver.h & pblib_solver.c which implements an interface with pblib compliant solvers (file based interface).

3.4.2 Criteria and combiner implementation

A flexible implementation of criteria and combiners has been introduced. Criteria are implementation of the abstract class abstract_criteria while combiners are implementation of the

abstract class abstract_combiner. Note that some combiners, like the agregate combiner, are implementation of both the abstract_criteria and the abstract_combiner classes. Such combiners can thus be either used as combiner or criteria. This architecture supports the flexible user defined criteria combination introduced in **mccs**.

Pure combiners are:

- lexicographic_combiner.h & lexicographic_combiner.c
- lexsemiagregate_combiner.h & lexsemiagregate_combiner.c
- lexleximin_combiner.h & lexleximin_combiner.c
- lexleximax_combiner.h & lexleximax_combiner.c

Classes which implement both combiner and criteria are:

- agregate_combiner.h & agregate_combiner.c
- lexagregate_combiner.h & lexagregate_combiner.c
- leximin_combiner.h & leximin_combiner.c
- leximax_combiner.h & leximax_combiner.c

Pure criteria are:

- removed_criteria.h & removed_criteria.c
- changed_criteria.h & changed_criteria.c
- notuptodate_criteria.h & notuptodate_criteria.c
- nunsat_criteria.h & nunsat_criteria.c
- new_criteria.h & new_criteria.c
- count_criteria.h & count_criteria.c

Chapter 4

PackUP

PackUP is a framework for solving the package upgradability problem. It translates the problem to a *weighted partial MaxSAT formula* [LM09] and uses a dedicated solver for the formula. The following are the options for solving the formula.

1. the MaxSAT solver msuncore
2. the optimization pseudo-Boolean problem (OPB) solver WBO
3. an external OPB solver, for example `minisat+`

The options (1) and (2) provide two distinct approaches to solving the problem. The option (3) should be seen as an open platform for other researchers trying their solvers on the upgradability problems. Moreover, PackUP together with `minisat+` provide a *free and open-source* implementation of an upgradability solver.

Figure 4.1 schematically depicts the possible workflows in `PackUp`. The input given in the format CUDF is encoded into a weighted MaxSAT formula by the tool `cudf2msu`. This formula can either be solved by the MaxSAT solver integrated in `cudf2msu`, or, by the solver `bmo-pblex`. Since `bmo-pblex` is a separate tool, which operates solely on the formula, a mapping between packages and variables is required to produce a solution in CUDF; the script `sol2cudf` reconstructs a solution for the package upgradability problem from a solution of the formula. Finally, when an external OPB solver is used, cudf2msu calls it repeatedly to obtain the solution.

The encoding of the problem is carried out in similar fashion to the encoding of the solvers presented in deliverable 4.2 [ALLM10]. However, some novel techniques were explored and these are described in section 4.2 using the notation introduced in the following section. section 4.3 discuss the techniques used for solving the weighted partial MaxSAT formula.

4.1 Preliminaries

`PackUp` supports CUDF 2.0 [TZ09b], but since the full description of CUDF 2.0 is beyond the scope of this report, only the prominent features of the format are considered. The following concepts and notations are simplifications of the formal semantics used in the CUDF 2.0 standard [TZ09b]. 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 as a partial function from name-version pairs to a tuple

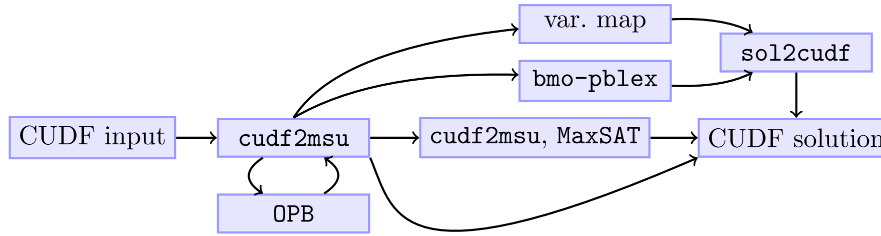

Figure 4.1: Workflows in PackUp

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 the version 1 of package p conflicts with the version 2 of package x and with all the versions of the package y except for the version 3. For simplicity, we assume that only $(y, \neq, v)$ is allowed to be a member of $\phi(p,v).\,\mathit{conflicts}$ when $p = y$, meaning that version v conflicts with all other versions of p.

The *depends* property is a conjunction of disjunctions of constraints whose satisfiability is the satisfiability of constraints extended by 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,

$$\phi(p,2).\,\mathit{depends} = ((x, \geq, 3) \wedge (y, \geq, 3)) \vee (z, \geq, 10)$$

means that version 2 of package p requires a version 10, or higher, of package z, or, packages x and y with versions 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 (l_i, l_d) where l_i is a set of constraints determining the packages that must be installed and l_d is a set of constraints determining 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 such that 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.

To introduce the optimization criteria we use several auxiliary definitions. We write $i_\phi(p)$ for the set of versions of a given package, We introduce a collection of measures of how much a solution ψ changes the original package universe ϕ: the number of packages removed, number of new packages, number of packages changed,

$$\begin{array}{lcl} i_\phi(p) & = & \{v \mid (p,v) \in Dom(\phi) \land \phi(p,v).\, installed = true\} \\ removed(\phi,\psi) & = & |\{p \mid i_\phi(p) \neq \emptyset \land i_\psi(p) = \emptyset\}| \\ new(\phi,\psi) & = & |\{p \mid i_\phi(p) = \emptyset \land i_\psi(p) \neq \emptyset\}| \\ changed(\phi,\psi) & = & |\{p \mid i_\phi(p) \neq i_\psi(p)\}| \\ notuptodate(\phi,\psi) & = & |\{p \mid i_\psi(p) \neq \emptyset \land v_{\max} \notin i_\psi(p)\}| \end{array}$$

In addition, *unmet-recommends*(ϕ,ψ) 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 the *removed*, *new*, *changed*, and *notuptodate*, e.g. $(removed, new)$. 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 under the lexicographic ordering. For instance, for the criterion $(removed, changed)$ a solution ψ_1 is better than ψ_2 iff

$$\begin{array}{ll} & removed(\phi,\psi_1) < removed(\phi,\psi_1) \\ \lor & (removed(\phi,\psi_1) = removed(\phi,\psi_1) \land changed(\phi,\psi_1) < changed(\phi,\psi_1)) \end{array}$$

4.2 Encoding

The encoding process is performed in the following sequence of steps.

1. *read in the problem*: reads the problem into dedicated data structures;
2. *slice*: traverses the data structures obtained in the previous step and discards all packages that are certainly unnecessary to provide a solution [TSJL07];
3. *encode package constraints*: captures conflicts and depends;
4. *encode request*: captures the given request;
5. *encode preference*: captures the given preferences;
6. *encode auxiliary variables*: generates additional formulas giving semantics to auxiliary variables used in the previous steps.

The encoding relies on propositional logic with the standard notions of *clause* being a disjunction of *literals* and a literal either a Boolean variable or its negation. To encode the problem we produce a weighted partial MaxSAT formula [LM09], which comprises two sets of clauses: *hard clauses* and *soft clauses* where each soft clause has a non negative weight. A solution to such 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 the weight W will be denoted as (W, c).

Whether a package p with version v is installed, 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* [TTKB09]):

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

Table 4.1: Definition of the operators $\mathcal{C}$ and $\mathcal{D}$ for constraints.

- $\mathrm{u}{\uparrow}_p^v$ — 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 4.1 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 $\mathcal{C}\left[x,(q,\mathit{relop},n)\right]$ always yields one or two clauses and that $\mathcal{D}\left[x,(q,\mathit{relop},n)\right]$ 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:

$$\mathcal{C}[x,l] = \bigwedge\nolimits_{r\in l}\ \mathcal{C}[x,r] \qquad \begin{aligned}\mathcal{D}[x,l_1\wedge l_2] &= \mathcal{D}[x,l_1]\ \wedge\ \mathcal{D}[x,l_2]\\ \mathcal{D}[x,l_1\vee l_2] &= \mathcal{D}[x,l_1]\ \vee\ \mathcal{D}[x,l_2]\end{aligned}$$

Apart from encodings of constraints between packages, we need to give the interval variables their intended meaning. To this end, we generate the following clauses for each package p and version v.

$$\begin{aligned} I_p^v \;=\; & (\neg\,\mathrm{i}{\uparrow}_p^v \vee x_p^v \vee \mathrm{i}{\uparrow}_p^{v+1}) \;\wedge\; (\neg\,\mathrm{u}{\uparrow}_p^v \vee \neg x_p^v) \;\wedge\; (\neg\,\mathrm{u}{\uparrow}_p^v \vee \mathrm{u}{\uparrow}_p^{v+1})\\ \wedge\; & (\neg\,\mathrm{i}{\downarrow}_p^v \vee x_p^v \vee \mathrm{i}{\downarrow}_p^{v-1}) \;\wedge\; (\neg\,\mathrm{u}{\downarrow}_p^v \vee \neg x_p^v) \;\wedge\; (\neg\,\mathrm{u}{\downarrow}_p^v \vee \mathrm{u}{\downarrow}_p^{v-1})\end{aligned}$$

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:

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

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

Algorithm 1: Solving the problem with an external OPB solver

```
1 ω ← constraints corresponding to hard clauses in the problem;
2 for i ← 1 to n do
3     (outc, s_i) ← Optimize(ω, Σ_j k^i_j * x^i_j);          // optimize for f_i
4     C_i ← f_i(s_i);                                         // compute i-th element score
5     if outc = false then
6         return false;                                       // the problem does not have a solution
7     ω ← ω ∧ (Σ_j k^i_j * x^i_j ≤ C_i);                      // fix the value of a solution
8 return s_n
```

capture the lexicographic ordering on the scores [Ehr05], 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* $(W_r, \mathrm{i}\Uparrow^1_p)$.

For the function *new*: *If $i_\phi(p) = \emptyset$ then generate the soft clause* $(W_n, \mathrm{u}\Uparrow^1_p)$.

For the function *changed*: *Let s_p be a fresh variable then generate the following hard clauses $\neg s_p \vee x^v_p$ if $\phi(p,v).installed = true$ and $\neg s_p \vee \neg x^v_p$ if $\phi(p,v).installed = false$; add the soft clause (W_c, s_p).*

For the function *notuptodate*: *Let t_p be a fresh variable; generate the hard clauses $\neg x^v_p \vee t^p$ for all $(p,v) \in Dom(\psi)$; generate the soft clause $(W_n, \neg t_p \vee x^{v_{max}}_p)$ where v_{max} is the maximal version of p appearing in ϕ.*

For the function *unmet-recommends*: *For each clause in $c \in \mathcal{D}(x^v_p, \phi(p,v).recommends)$ generate the soft clause (W_u, c).*

4.3 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 the large weights resulting from the lexicographic ordering [ABL+10]. Hence, `PackUp` uses solvers that are specially adapted to lexicographic optimization [ALMS09]. `cudf2msu` contains an extension of the solver `msuncore` and `bmo-pblex` is an extension of the solver `WBO`. The solver `msuncore` searches on the *lower-bound* of the optimization function [FM06, MMSP09] and therefore is suitable for problems where the optimum is not too far from the best theoretically result. In contrast, `WBO` searches on the *upper-bound* [BF98] and therefore is expected to perform well on problems with

high deviation from the best theoretical result.

When PackUp is used with an external OPB solver, it invokes the solver multiple times using the techniques from [ALMS09]. The weighted partial MaxSAT formula representing the upgradability problem is converted to OPB using standard conversion techniques [MMSP09]. Then, for a criterion $(f_1, \ldots, f_n)$, each f_i corresponds to a sum of the form $\Sigma_j k^i_j * x^i_j$ for some integer coefficients k^i_j and variables x^i_j. line 1 presents in pseudo-code how the external OPB solver is called. PackUp maintains a set of constraints ω, which are gradually strengthened. For each function in the criterion it calls the OPB solver (line 1). In the pseudo-code, the solver is represented by the function Optimize(τ, ξ) where τ is an OPB constraint and ξ is a function to be optimized. The return value is a pair $(\texttt{outc}, s)$ where outc represents whether the problem has a solution or not, and, s represents a solution if one was found. Once the optimum for f_i is found, its value is fixed (line 1) and the algorithm moves onto the function that follows in the lexicographic ordering.

Chapter 5

Constraint programming and the package installability problem

This chapter presents the approach we used to solve the package installability problem by using constraint programming. It is divided into two main sections, the first one describing the process of taking a CUDF file and interpreting it and the second one describing the solving process itself: the modelling of the problem as a set of constraints and their semantic.

In the first section, we realized that a CUDF specification was still high level from a solver perspective. This is based on the fact that a big part of the solver process was dealing with *interpreting* the specification. As a result we decided to isolate and modularize this part and to establish an intermediate low level language that was easier to interpret for the solver. This language is KCUDF which stands for *Kernel CUDF*. It presents a simplified semantics still preserving the complete problem information. A set of tools were also developed in order to convert from and back to CUDF specifications.

The second section presents the constraint-based solver. After analyzing the problem, and the possible different ways in which constraint programming existing research could be used to solve, it was decided to use constraints on binary relations. This decision is based on two aspects: first, the core of the problem can be expressed (in natural language) as relations among packages. Second, the opportunity to compare how different approaches, that the research area offers, can tackle the problem and solve it.

5.1 The solver infrastructure

5.1.1 KCUDF

CUDF is a great achievement promoted by this project and as such, it offers great compatibility and a common base to work in applications that rely on package information. However, the solvers by themselves work at a lower level of abstraction and a CUDF specification still needs some interpretation before being an input to a solver. This is why we propose KCUDF as a kernel language that is more suitable for the solvers.

The language is defined by the following grammar:

```
KCUDFSpec ::= <PkgSpec> <RelSpec>
```

```
PkgSpec ::= 'P' <Id> <Keep> <Install>

RelSpec ::= <DepSpec>* <ConfSpec>* <PvdSpec>*

DepSpec ::= 'D' <Id> <Id>

ConfSpec ::= 'C' <Id> <Id>

PvdSpec ::= 'R' <Id> <Id>

Keep ::= 'K' | 'k'

Install ::= 'I' | 'i'

Id ::= [0-9]+
```

Every line in a KCUDF specification is either declaring the existence of a package or defining the relations among them. When a package is defined, the first character of the line contains the letter `P` followed by a numerical identifier and two letters specifying how the package must be treated by the solver regarding its presence in the system. `K` stands for *Keep* and its uppercase form indicates the solver, that whatever the installed state of this package is, has to remain at the end of the process. `I` represents an *installed* package while the same in lower case would represent that the package is not installed.

For instance, the statement `P 42 K i` will be interpreted by the solver as: the package with identifier 42 will be kept uninstalled from the system. In a similar way, `P 54 K I` will instruct the solver to install the package.

Additionally, statements starting with `D` represent the existence of a dependency relation between the two packages specified (the first one depending on the second one). In the case of a `C` a conflict relation between the two mentioned packages is defined. Finally, statements starting with `R` define that the first package provides the second one.

The benefits of having KCUDF as the input language for the solver are:

- No interpretation has to be performed at the solver level.
- The information can be parsed by a single traversal of the input file.
- The Installability problem is encoded in the install and keep attributes on the packages.

One of the most important drawbacks is that it does not offer a common way to encode the optimization criterion. However, it is extensible: new attributes can be defined in a per-package basis and a lexicographical search criterion can be expressed on them.

Apart from defining this format we also provide some tools that facilitate working with KCUDF files:

- A program called `cudf2kcudf` that takes a CUDF specification and translates it to KCUDF.
- `kcudf-reduce` which takes a KCUDF and outputs an equivalent problem (in KCUDF) that discards the packages that will not affect the solution of the solver.

This tool is fundamental in the case of our constraint based solvers because it eliminates the noise caused by packages that does not have an effect on the solution.

- `kcudf2cudf` that takes a KCUDF and an original CUDF specification and produces a CUDF. In this CUDF the packages have the state reported by the KCUDF specification. This program is useful to report solutions.

This set of tools allows a clean integration of KCUDF-based tools with others such as `libcudf` which is provided by other partners of the project and provides parsing functionalities on CUDF specifications.

Figure 5.1 presents a graphical description of the interaction between the proposed components. The modules *Translator*, *Reducer* and *Formatter* are directly supported by the applications *cudf2kcudf*, *kcudf-reduce* and *kcudf2cudf* presented above. An important aspect of the presented infrastructure is that the communication between modules is done via KCUDF. The reducer module provides two outputs: a description with the packages and their corresponding states in the system that does not have to pass through the solver (tagged with E in the figure); and the equivalent sub-problem that needs to be solved. At the end, both solutions emitted by the solver and the output of the reducer are mixed by the formatter to produce a CUDF output to the installability problem.

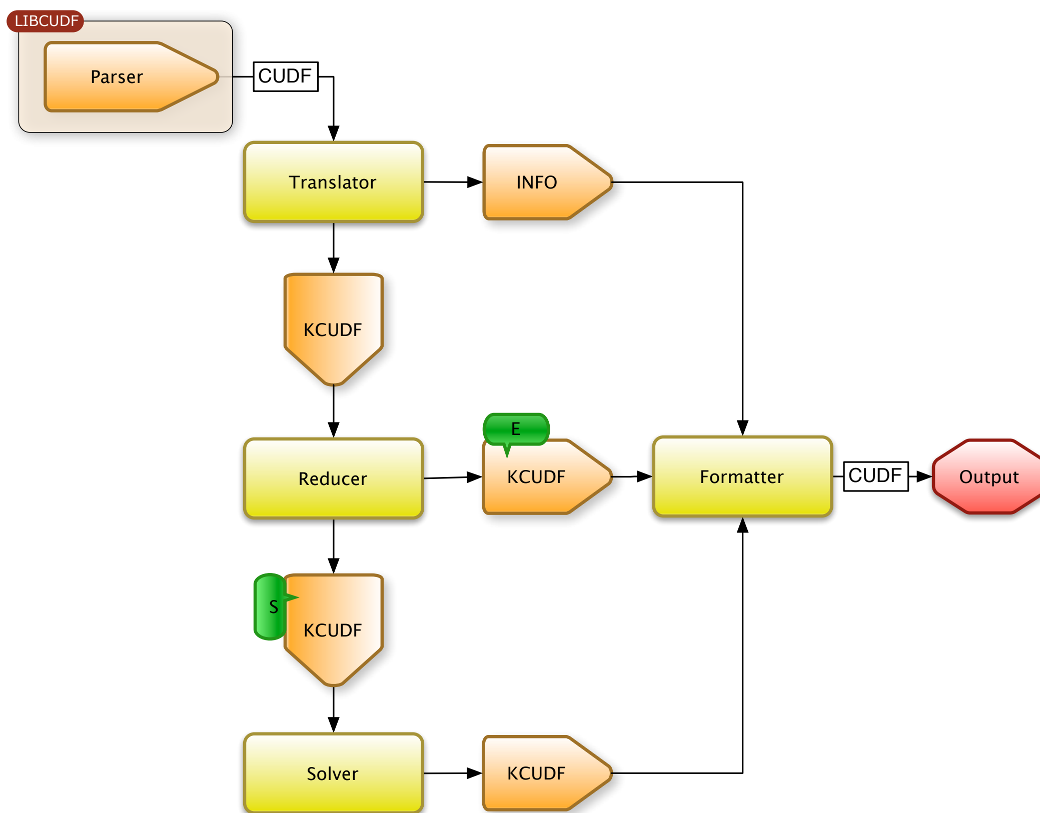

Figure 5.1: Solver infrastructure and supporting components.

5.1.2 Reduction process

The process of reducing the input to only consider packages that need to be taken into account by the solver is crucial for the kind of solver we propose. The solving process uses a considerable amount of resources and its complexity depends on the domains of the constraint variables (this topic will be described in depth in the next part of this report). The reducer aims at trimming the input taking into account several aspects of the problem:

- User requests usually specify small changes compared to the set of available packages in the universe.
- Only packages marked to be kept installed or uninstalled need to be considered in addition with the packages related in some way with the packages in the request.

For these two reasons, there are a considerable number of packages that are not going to change due to the solving process. This is the set of packages that is extracted by the reducer algorithm. This algorithm works on the assumption that each package has an initial state (the first time it is considered by the reducer) and this state can change according to the diagram presented in Figure 5.2.

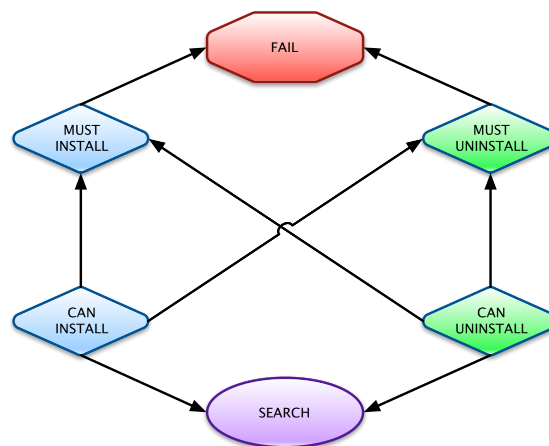

Figure 5.2: Reducer state diagram.

The states labelled *CanInstall (CI)* and *CanUninstall (CU)* correspond to packages that, have `k I` and `k i` in the KCUDF specifications. Packages in *MustInstall (MI)* and *MustUninstall (MU)* correspond to `K I` and `K i`. During the reducing phase, packages change from one state to another, taking into their relations with other packages. For instance, if a package needs to be installed, it will get in the state *MI* and all its dependencies will have this state too. On the other hand, all its conflicts will go to state *MU*.

There are two other states that are considered by the reducer: a state labelled *Search (SR)* that contains part of the packages for which the solver needs to provide a solution. The state *Fail* indicates a failure or inconsistency detected by the reducer. In this case the reducer has found that the request and the current information make the problem unfeasible.

The reduction algorithm will apply the rules introduced by the relations among packages until no change of state is possible. At this point if there is any package in the failure state, the problem is known to have no solution. For all the packages that end up in the state *SR* the

solver has to find a solution. Enough information about this packages is provided in the output of the solver.

5.2 Constraint based solver for the PIP using binary relations

In the previous section we described the set of tools using around the solver component. This section is completely devoted to the solver itself and presents the way we solve the problem using constraints on binary relations.

Binary relations match perfectly with the existing relations among packages: *Dependencies*, *Conflicts* and *Provides*. In this part of the report we show how they are directly expressed in terms of decision variables that have relations as domain.

5.2.1 Constraint model

The problem data can be naturally described in terms of relations among packages. This fact can be further exploited at the solving process by exploiting the relation concept to solve it. To do this, the constraint model includes variables with relation domains and constraints on them.

Variables

- *Inst*: The goal of the solver is to find a set of packages to be installed that fulfills the user request and this is exactly what the *Inst* variable represents. It is a set variable with domain:

  $$MI \subseteq Inst \subseteq U \setminus MU$$

  where U is the set of all possible packages defined by the number of repositories accessible from the user system.

- *Dep* represents the dependency relation, its domain is:

  $$Dep_0 \subseteq Dep \subseteq Dep_{max}$$

  The argument for having the dependencies as variables in our model is that new dependencies can be inferred by constraint propagation. For instance, only direct dependencies are given by the input. Dep_{max} is computed with the information in the input. Basically when a package p depends on a virtual package v, the dependencies of v are considered as *possible* dependencies of p. Note that this property is transitive.

- *Conf* represents the conflict relation, its domain is:

  $$Conf_0 \subseteq Conf \subseteq Conf_{max}$$

  In the same way as new dependencies can be inferred by constraint propagation it is possible to do the same for the conflicts. $Conf_{max}$ is computed with the information of the input and a similar process as the one described for Dep_{max}. However in this case it is not a transitive property.

- *Prov* represents the provides relation. It is a binary relation with domain:

$$\emptyset \subseteq Prov \subseteq Prov_0$$

This is the only relation variable with an empty lower bound. The reason for this is that only when packages get installed the variable will change with the provided virtual packages.

The argument for considering the relations as variables and not just as raw input data is that during the solving process new information can be inferred. At first sight this looks like something that is not possible since package relations are there from the very early stages of their design, however, the request and particular user configurations can change this aspect dramatically. Just to give an example, suppose that installing package p requires some functionality q that is provided by packages r and s. As soon as one of the packages r or s become uninstallable for any reason, a new element of the dependency relation emerges, this time between p and the remaining package.

The current installation plays no role in the domains of the variables and will not be used for the constraints either. This guarantees that the full search space is considered without artificial restrictions and that any inconsistency in the current installation will not propagate to the solution. By contrast, we will use the information about the current installation to guide the search and to evaluate the quality of solutions.

Constraints

To enforce the conflict relation the following constraint imposes that two conflicting packages cannot be installed together.

$$\text{conflicts}(Conf, Inst) \equiv \forall x, y : x \in Inst \wedge (x, y) \in Conf \implies y \notin Inst$$

The constraint enforcing the dependency relation among installed packages is presented below. It ensures that the dependencies of an installed package are installed. Conversely, if a package p depends on a package q that will not be installed, p will not be installed as well.

$$\text{dependencies}(Dep, Inst) \equiv \forall x, y : x \in Inst \wedge (x, y) \in Dep \implies y \in Inst$$

The following constraint states that any installed package must have a provider. If an installed package has a unique candidate provider left, the pair representing this becomes part of *Prov*. Conversely, when a package is marked as not installed, all the pairs leading to it can be excluded from *Prov*.

$$\text{provides}(Prov, Inst) \equiv \forall x : x \in Inst \implies \exists y \in Inst \wedge (y, x) \in Prov$$

Concrete packages (the ones that really install files on the computer) are self-providing. Virtual packages are not. Generally, *Prov* is a subset of *Dep* since if concrete package p provides

functionality q, it is impossible to install p without also having q available. Because of the inherent disjunction that it represents, the implementation of this constraint only propagates when a package has only one provider or no provider at all.

These constraints do not exploit much the fact that we have relation variables and not values. Redundant constraints will be added later and will provide further propagation and pruning of the search space.

5.2.2 Traversal of the search space

The constraints previously presented are enough to represent and solve the PIP. However, because of the search strategy we need to add one more constraint to the problem.

The natural search strategy would be to add basic constraints to *Inst* as it is the variable whose value is the result we are searching for. However, there is no natural criterion to decide on which package to include or exclude at any point in the search. A far better approach is to decide to include or exclude a potential provider of a package that is known to be installed in the solution, but which as not yet a known installed provider. Packages that are not installed do not need a provider. By adding the following constraint, we can simply search on *Prov* and have *Inst* be determined as a side-effect. Note that the decisions will be taken on *Prov* and will be propagated to *Inst* by this constraint.

$$\text{Image}(\mathit{Inst}, \mathit{Prov}) \equiv \forall x, y : (x, y) \in \mathit{Prov} \implies y \in \mathit{Inst}$$

Adding a pair to *Prov* will include the *provided* package in *Inst*; removing *all* pairs leading to a package, will remove it from *Inst*. Therefore, when *Prov* is determined, *Inst* is also completely known. The propagation between *Inst* and *Prov* is bidirectional.

Here is the branching algorithm. The choice point is created as described above by including or excluding a pair from *Prov*. That leads to a package that is known to be installed and not provided by an installed package. Which candidate provider gets selected is decided by the `selectProvider` procedure that allows the use of different criterion to compute the provider. In this algorithm `ProvI` stands for the *provider* relation (i.e. Prov^{-1})

```
branch(ProvI, Inst) {
 forall (p in domain(ProvI) :
        Inst.contains(p) && Prov.minCard(p) == 0) {
    v = selectProvider(p,ProvI);
    try ProvI.include(p,v) | ProvI.exclude(p,v);
  }
}
```

So far we have presented the constraints used to solve the PIP. From the problem perspective they are enough to bring correct solutions. Note that these are not necessarily the only constraints we use to solve the problem. The next section presents redundant constraints that are considered as part of the model.

5.2.3 Redundant constraints

The constraints presented so far are not taking much advantage of the fact that we have relational variables. However, many deductions can be done that will increase propagation and pruning and therefore reduce the need for search.

A first batch of redundant constraints derives from the fact that if p depends on q, any installation in which p is present will necessarily also include q. Because of this, any dependency of q can be seen as a dependency of p and any conflict of q is a conflict of p. These properties are translated into two constraints. The first one ensuring that *Dep* is transitively closed. While the second enforces that *the conflicts of the dependencies of a given package are also conflicts of the package itself.*

$$\text{TransitivelyClosed}(\mathit{Dep})$$
$$\text{Follow}(\mathit{Dep}, \mathit{Conf}, \mathit{Conf})$$

These two constraints are effective but stop at the same point: virtual packages have generally no dependencies or conflicts. Their providers do, but because of the disjunction that the virtual package represents the conflicts and dependencies of a provider cannot be safely considered as conflicts and dependencies of the virtual package itself.

However, the providers of a virtual package are generally different versions of a same program or several similar programs. It is common for them to have the same dependencies and conflicts. And if *all* the candidate providers of a virtual package have a conflict or dependency in common, then that conflict or dependency can safely be considered as applying to the virtual package itself. In essence, this is a specific form of constructive disjunction. This could be represented in the model by the following constraints.

$$\text{FollowAll}(\mathit{Prov}^{-1}, \mathit{Dep}, \mathit{Dep})$$
$$\text{FollowAll}(\mathit{Prov}^{-1}, \mathit{Conf}, \mathit{Conf})$$

Informally, *FollowAll* captures the notion of transitivity between two relations to deduce more information about a third one. For instance, it allows us to reason about packages in the *provides* and in the *dependency* relation to possibly deduce new dependencies: when all the providers of a package share the same dependency then this becomes a dependency of the package itself.

The *FollowAll* constraint brings some implementation challenges that in some cases render it unpractical. The FAAA constraint overcomes this problem at the price of an extra argument. It still captures the notion of transitivity but this time limiting it to the elements that appear in its first argument. For our problem we use Prov_0 (only the ones that have a provider) to limit the action of the constraint instead of considering all the packages in the repository.

$$\text{FAAA}(\mathit{Prov}_0^{-1}, \mathit{Prov}^{-1}, \mathit{Dep}, \mathit{Dep})$$
$$\text{FAAA}(\mathit{Prov}_0^{-1}, \mathit{Prov}^{-1}, \mathit{Conf}, \mathit{Conf})$$

By Prov_0^{-1} we mean the inverse relation: Prov_0 denotes the *provides* relation and Prov_0^{-1} the *provider* one.

Heuristics

The definition of `selectProvider` in Section 5.2.2 is purely heuristic, it could be for instance selecting the first provider available:

```
int selectProvider(p, ProvI) {
  return ProvI(p).begin();
}
```

In this implementation the first possible provider of `p` is returned. This heuristic does not lead to good solutions for the user. When a user installs a package it is expected that solutions describe installations that are close to the current system state. A solution that calls for the removal of many packages might be a solution to the problem but will not be acceptable for the user. Taking into account the current setup in the heuristic will help provide better solutions. This is done by:

```
int selectProvider(p, ProvI, Inst0) {
  set<int> c = intersect(ProvI(p),Inst0);
  if (c.empty()) return ProvI(p).begin();
  else
    return c.begin();
}
```

This time the heuristic checks for providers of `p` already present in the initial installation (`Inst0`) and considers them first. If there is no provider installed, any package that can provide `p` is selected. Even if the initial installation is not consistent for the reasons mentioned in Section 5.2.1, we are not risking correctness as this information is only used as an heuristic.

When there are no possible providers of `p` that are part of `Inst0` there is still an heuristic to apply. We can associate a rank to every package given how difficult it is to install. This difficulty can be measured by the number of conflicts it has and by the number of dependencies. We can consider packages with bigger number of conflicts and dependencies as packages that should be installed first and then get more pruning of the search space. The following heuristic implementation integrates this concept:

```
int selectProvider(p, ProvI, Dep, Conf, Inst0) {
  set<int> c = intersect(ProvI(p),Inst0);
  if (c.empty())
    return ProvI(p).begin();
  return difficult(p, ProvI, Dep, Conf);
}
```

The implementation of `difficult` is straightforward, it just consists of traversing all the providers of `p` and returning the one for which the number of dependencies and conflicts is maximal. Rather than considering static information, this heuristic will consider the current state of the dependencies and the conflicts. That is, it will consider dependencies and conflicts that were deduced.

5.2.4 Optimization

Instances of PIP are typically under-constrained and have many solutions. We use heuristics to provide user-acceptable solutions but this is generally not enough. In most cases an optimization

criterion can be defined. Different users will have different ideal functions. As an example, we can define the paranoid criterion for users who want to avoid removing currently installed packages as much as possible.

To express this criterion, we introduce two new variables. The first is a set, defined as the intersection of the solution installation with the actual installation. The other is an integer and is simply defined as the cardinality of the first one.

$$\text{Common} = \textit{Inst} \cap \text{Inst}_0$$
$$\text{Close} = \text{card}(\text{Common})$$

Of course the optimization criterion and the heuristic have to be closely matched to have good performances. However, even if each user has a different view on the prefect solution, the basic ideas are common. In the optimization algorithm below, `BestSF` represents the best solution found so far and `Curr` represents the current search space.

```
void optimize(BestSF, Curr) {
  Curr.Close > BestSF.Close;
}
```

5.2.5 Current results

To measure the performance of the implemented solver we use data from the Mancoosi project. This project aims at integrating new technologies in FOSS distributions. In particular, it focuses on the development of efficient algorithms and tools to plan system upgrades based on various information sources about software packages and optimization criteria. Part of this project consists of a solver competition where different technologies are used to tackle the PIP.

The experiments we have done with instances of the solver competition provide evidence that our approach to tackle PIP can be competitive. This section presents a description of the data used and the times we got with our solver.

The data consists of a universe from a Debian distribution using stable, testing and unstable branches. On some of the instances a random package is selected to be installed (this is where the instance name cameos from). The state of the packages is inconsistent. This means that the installation is broken and that the solver will have to fix it. As an optimization criterion a system closer to the current is requested.

Table 5.2.5 shows the name of the instance, the time used by the solver to find a first solution (TFFS) using as heuristic the installation of the packages already present in the system. The third column presents the time for the solver start proving optimality (TOP), the fourth column presents the number of solutions found when the search was stopped (Sols.) and finally the last column presents the time spent by a SAT based solver to solve the problem. All times are in seconds.

Due to the size of the search space our solver is not able to finish in a practical time (e.g. 5 minutes is the maximum time in the solver competition). The time to find the first solution is around 8 minutes (average). In most of the benchmarks it takes 2 minutes more to find about 30 solutions to the instance. After that time the number of failures start increasingly very fast. Because of this, and the way the search tree looks like we think that the huge amount of failures

Instance	TFFS	TOP	Sols.	SAT
ark	408	470	27	282
codebreaker	400	502	32	282
dpkg-dev	408	507	30	285
libnss3-ld	410	495	29	289
libgnokii4	413	510	33	283
mercurial-common	411	521	30	292
mono-gac	411	535	31	291
openoffice.org-l10n-da	411	519	29	284

is due to the fact that the solver is traversing the search space proving optimality. In fact the last solution found differs only by 70 (average) packages from the initial one. The fact that the initial installation is inconsistent could imply that those packages cannot be installed. It is important to highlight that the SAT solver is able to proof the optimality of the solution.

In conclusion, to reduce the time to check the optimality of a solution we need to improve the complexity of domain operations (by a profiler analysis, most of the time is consumed by the execution of domain iteration done by Follow and FAAA). We wrongly believe that a better domain representation will change the way constraints modify the domain and will leaf to better times. The times presented in this section were taken on an explicit domain representation. These times are an argument to design the new domain representation. For space reasons that part of the work is not presented here but it is available on the Mancoosi repository[1].

[1] https://gforge.info.ucl.ac.be/svn/mancoosi/trunk/PIPsolvers/ccp-kcudf/docs/confirmation.pdf

Bibliography

[ABL$^+$10] 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*, volume 29 of *EPTCS*, pages 11–22, 2010.

[ALLM10] Josep Argelich, Inês Lynce, Olivier Lhomme, and Claude Michel. First version of the Mancoosi specialised cudf solver plugin for the modular platform manager. Deliverable 4.2, The Mancoosi Project, February 2010. http://www.mancoosi.org/reports/d4.2.pdf.

[ALMS09] Josep Argelich, Inês Lynce, and João P. Marques-Silva. On solving boolean multilevel optimization problemse. In Craig Boutilier, editor, *IJCAI*, pages 393–398, 2009.

[BF98] 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.

[BL09] Sylvain Bouveret and Michel Lemaître. Computing leximin-optimal solutions in constraint networks. *Artif. Intell.*, 173(2):343–364, 2009.

[Ehr00] M. Ehrgott. *Multicriteria optimization*. Lecture Notes in Economics and Mathematical Systems. Springer-Verlag, 2000.

[Ehr05] Matthias Ehrgott. *Multicriteria Optimization (2. ed.)*. Springer, 2005.

[FM06] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In Armin Biere and Carla P. Gomes, editors, *SAT*, volume 4121, pages 252–265. Springer, 2006.

[LM09] 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.

[MBC$^+$06] 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'2006,21st IEEE/ACM International Conference on Automated Software Engineering*, pages 199–208. IEEE Computer Society, 2006.

[MMSP09] 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.

[Ogr97] Wlodzimierz Ogryczak. On the lexicographic minimax approach to location problems. *European Journal of Operational Research*, 100(3):566–585, 1997.

[TSJL07] Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. OPIUM: Optimal package install/uninstall manager. In *ICSE*, pages 178–188. IEEE Computer Society, 2007.

[TTKB09] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.

[TZ09a] Ralf Treinen and Stefano Zacchiroli. Common upgradeability description format (CUDF) 2.0. Technical Report 3, The Mancoosi Project, November 2009. http://www.mancoosi.org/reports/tr3.pdf.

[TZ09b] Ralf Treinen and Stefano Zacchiroli. Common upgradeability description format (CUDF) 2.0. Technical Report 003, MANCOOSI, November 2009.

[Yag88] R. R. Yager. On ordered weighted averaging aggregation operators in multicriteria decisionmaking. *Systems, Man and Cybernetics, IEEE Transactions on*, 18:183–190, 1988.