mancoosi
managing software complexity

First version of the Mancoosi specialised CUDF solver plugin for the modular platform manager

Deliverable 4.2

Nature : Deliverable
Due date : 1.2.2010
Start date of project : 01.01.2008
Duration : 36 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	Josep Argelich <jargelich@diei.udl.cat> Inês Lynce <ines@sat.inesc-id.pt> Olivier Lhomme <Olivier.Lhomme@fr.ibm.com> Claude Michel <Claude.Michel@i3s.unice.fr>
Internal review	Roberto Di Cosmo <roberto@dicosmo.org> Anne-Sophie Réfloc'h <anne-sophie.refloch@univ-paris-diderot.fr>
Workpackage number	WP4
Deliverable number	2
Document type	Deliverable
Version	2
Due date	01/2/2010
Actual submission date	21/5/2010
Distribution	Public
Project coordinator	Roberto Di Cosmo <roberto@dicosmo.org>

Contents

1 Introduction

Deliverable D4.2 is a prototype, that can be downloaded from the URL http://www.mancoosi.org/software/D4.2.tar.gz.

This report accompanies the D4.2 prototype.

It provides an introductory insight of multicriteria optimization in general and multicriteria optimization issues related to upgradeability problems.

It also provides additional information about the Mancoosi plugins that are found in the prototype and on the internal competition, and used to perform a preliminary evaluation of the different plugins.

Multicriteria optimization is a corner stone of the Mancoosi approach to upgradeability issues. Indeed, an upgradeability problem might have a significant amount of solutions to suit, more or less, to users' needs. To deal with this issue, the Mancoosi project relies on multicriteria optimization. The section 2 introduces the basics of multicriteria in the context of upgradeability problems.

2 On solving the upgradeability problem

2.1 Mathematical model

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 implemented as constraints.
- The quality of solutions to upgradeability problems are measured using objective functions

The upgradeability problem boils down to solving a combinatorial optimization problem: find values for the variables maximizing the criteria and satisfying the constraints.

In general, we have several quality measures of solutions to the upgradeability problem. For example, we may want to minimize the download time, but we wish to get the most recent version of packages and we, also, want to minimize the disruption of the current configuration. In general, those different measures are contradictory: clearly, on an old installation, we cannot simultaneously minimize the download time and maximize the freshness of packages' versions. That is to say that the upgradeability problem, generally, is a multicriteria optimization problem.

Before discussing the general multicriteria issue, it is necessary to determine how easy are the two main subproblems: the simple satisfiability of the upgradeability problem, and the mono-criterion optimization upgradeability problem.

First of all, the satisfiability of the upgradeability problem, i.e. without taking into account equality measures of a solution, is already theoretically very difficult. It is an NP complete

problem: the typical number of variables is 30000, the size of the space to explore is potentially 2^{30000} and some instances of the problem may be well beyond the capacity of the computers. In practice, fortunately, this problem is much easier than expected by this theoretical view: the density of solutions is high, and every instance we get in the experiments was solved in a few minutes.

Given a single criterion, the problem becomes much more difficult to solve. This can be simply explained: even if the density of a solution is high, the density of optimal solutions may be quite low. Nevertheless, in practice, the experiments done in Mancoosi pointed that the optimal solution is reachable by some solvers in a few minutes. Anyway, even when the optimal solution needs too much CPU time to be found, a user is generally satisfied with a *good* solution which is non optimal, but that may be quite easier to find.

2.2 Upgradeability as a multicriteria optimization problem

Now, it is possible to focus on the real problem, where several contradicting measures are used, giving several criteria. The problem is not only to find a good or an optimal solution. Indeed, before solving the problem, it is needed to define what kind of solution we want. Taking up the above example on an old installation, we are considering two criteria: minimize the download time, and maximize the freshness of the packages versions. We need to decide what kind of trade-off between the two criteria we want to have.

Lexicographic optimality A first compromise exemple is to favour any improvement of one criterion, even a very small one, to any improvement of the second criterion, even a huge one. 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 if it is equivalent for criteria 1..i-1, and is better for the i-th criterion This is the approach followed for the solvers' development and for the solvers' comparison this year. This approach has a significant advantage: it does not need the criteria's values to be compared. Indeed, the comparison of the values of different criteria is a difficult task, and can be specific to one user and one problem.

Pareto optimality Another approach, avoiding the difficult task of defining how the values of different criteria can be compared, is simply to generate all the solutions that may be preferred. A solution may be favoured, 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 if, for each criterion, s_1 is at least equivalent to s_2, and there exists a criterion such that s_1 that is strictly better than s_2. Intuitively, it is not possible to improve a pareto-solution criterion without a loss for another criterion.

In practice, the set of pareto-solutions is important, and the usual approach taken is to compute an approximation of this set.

Utility functions The comparison of the different criteria values is needed in all multicriteria methods to define what is a good trade-off. The lexicographic approach, indeed, defines the trade-off as no possible trade-off, and therefore does not need such a comparison. In general, this comparison is made through 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 the comparison between different possible 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 let v_1, v_2 be the values of z_1, z_2 in a given solution. Thus, we know that, in this solution, z_1 is more favoured than z_2 when $u_{z_1}(v_1) > u_{z_2}(v_2)$.

Trade-offs between criteria 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 preferred to s', noted $s >_m s'$. Many different preference orders $>_m$, and many corresponding multicriteria methods exist. For the upgradeability problem, the efforts are focusing on some 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 satisfying. 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 solution, many good solutions are not reachable. Furthermore, its sensitivity to the weights is very high.

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

This approach, unfortunately, produces many indistinguishable solutions, and some of them are not pareto-solution.

A common principle for more sophisticated approaches An idea,frequently encountered in advanced methods, is to order the criteria depending on the values they take in a solution. We give a detail of two of these methods.

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.

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 Leximax solution is simply a lexicographic solution on $p_1, ..., p_k$.

Leximin/leximax have nice properties, that make their outcomes intuitive for the user:

- they produce only pareto-solutions;
- they are *egalitarian* : a leximax solution is always a solution for the method maximizing the minimum;

- they verify the property of *reducibility*: if a criterion is set to a value that is found in an optimal solution, then leximin/leximax sill produce the same set of optimal solutions as without that criterion

3 The Mancoosi plugins

The complexity of the upgradeability problem is at least NP complete, and modern operating systems contain a huge number of packages (often more than 20 000 packages in a Linux distribution). Solving the upgradeability problem relies on complex algorithms whose efficiency is hard to predict. Instead of sticking to a given algorithm to solve the Mancoosi problem, another approach was implemented, where different possibles algorithms and solvers have been explored. This should improve our ability to handle the upgradeability problem while providing a broader experience on the right way, and if any, to solve it.

At this stage of the project, four different systems are under development:

- Apt-pbo developed by Caixa Magica relies on a pseudo boolean solver and benefit from the environment of a Linux distribution.
- INESC solver relies on an original approach based on MAXSAT.
- UCL solver relies on graph constraints.
- UNSA solver relies on Integer Linear Programs.

This strategy should offer a wide spectrum of approaches to handling the upgradeability problem, and maximize the quality of the results.

4 CUDFsolver

CUDFsolver represents UNSA efforts to implement a CUDF problem solver. It is written in C++ and relies on Flex and Bison to implement a CUDF reader. This solver translates the CUDF problem into a MILP problem and then relies on an underlying MILP solver to get a CUDF problem solution.

This section describes the MILP translation of the CUDF problem, as well as, the solver implementation.

4.1 A MILP model for the upgradeability problem

The upgradeability problem aims at finding the best solution according to some given criteria. That is why we investigated the capabilities of MILP to solve this problem. In other words, we translate the upgradeability problem into a minimisation problem of a set of binary variables under some integer linear equations and inequalities. Multicriteria optimisation is handled either through a classical implementation of lexicographic multicriteria optimisation or through an aggregate function.

4.1.1 Constraints

Modelling the CUDF problem as a linear integer program is quite straightforward. Each unique couple $< package, version >$ is represented by a binary variable, the value of which states whether the couple $< package, version >$ is installed or not. So, the domain of these variables is restricted to $\{0, 1\}$ and solving the problem consists in finding an assignment of these variables, i.e., determining whether the corresponding couple $< package, version >$ is installed or not in the final configuration.

A depend field provides a description of the related package dependencies by means of a conjunction of disjunctions of package names. Assume that p_v, i.e., package p in version v, has the depend field:

$$\bigwedge_{i=1}^{n} p_i\_v_i \wedge \bigwedge_{j=1}^{m} \bigvee_{k=1}^{l_m} p_{j,k}\_v_{j,k}$$

We translate such a formulae into a set of of integer linear inequalities in two steps:

1. The first set of conjunctions of the formulae is translated into the following inequality:

$$-n * p\_v + \sum_{i=1}^{n} p_i\_v_i >= 0$$

   Such an inequality ensures that all $p_i\_v_i$ are installed if $p\_v = 1$, i.e., if p_v is installed. Of course, the $p_i\_v_i$ can take any value if p_v is not installed.

2. The following integer linear inequality is generated for each disjunction:

$$-p\_v + \sum_{k=1}^{l_m} p_{j,k}\_v_{j,k} >= 0$$

   This inequality ensures that at least one of the $p_{j,k}\_v_{j,k}$ is installed if p_v is installed.

```
package: car
version: 1
depends: engine, wheel, door, battery
installed: true
description: 4-wheeled, motor-powered vehicle

package: gasoline-engine
version: 1
depends: turbo
provides: engine
conflicts: engine, gasoline-engine
installed: true

package: gasoline-engine
version: 2
provides : engine
conflicts : engine , gasoline-engine

package: electric-engine
version: 1
depends: solar-collector | huge-battery
provides : engine
conflicts : engine , electric-engine

package: battery
version: 3
provides : huge-battery
installed : true

...

request :
install : bicycle , electric-engine = 1
upgrade: door
```

Figure 1: A CUDF example (extracted from [TZ09])

Each conflict field is translated into the following inequality:

$$n' * p\_v + \sum_{p'\_v' \in Conflict(p\_v)} p'\_v' <= n'$$

where $Conflict(p\_v)$ is the set of packages conflicting with p_v and n' is the cardinality of $Conflict(p\_v)$.
Such an inequality ensures that none of the p_v conflicting packages can be installed if p_v is installed.

To illustrate this translation process, we provide hereafter part of the model generated for `gasoline- engine_1`, the gasoline-engine package in version 1 (see figure 1). `gasoline-engine_1`

depends from package `turbo_1` which only exists in version 1. To ensure that `turbo_1` is installed whenever `gasoline-engine_1` is installed, the following constraint is generated:

```
- gasoline-engine_1 + turbo_1 >= 0
```

`gasoline-engine_1` conflicts with any other version of gasoline-engine (like `gasoline-engine_2`) as well as with any other package providing the engine feature (like `electric-engine_1` and `electric- engine_2`). To ensure that none of these packages is installed whenever `gasoline-engine_1` is installed, the following constraint is generated:

```
3 gasoline-engine_1 + gasoline-engine_2
                    + electric-engine_1 + electric-engine_2 <= 3
```

The provide field does not directly involve constraint generation. As a matter of fact, it is taken into account while managing the depend or conflict fields through the interpretation of feature names into set of related package names. For instance, when package `car` asks for engine in its depend field, the set {gasoline-engine version 1, gasoline-engine version 2, electric-engine version 1} is substituted to engine.

Once constraints for all the versioned packages have been generated, the solver handles the problem requests. Install or remove requests are directly translated by a variable setting corresponding to the required status in the final configuration. For instance, constraint $p\_v = 1$ is generated for request `install: p = v` and constraint $p\_v = 0$ for request `remove: p = v`.

An upgrade request must ensure that only one version of the upgraded package will be installed and that the installed package version will be higher or equal to any installed version of the current package in the initial configuration. For instance, assume that gasoline-engine has 5 versions ranging from 1 to 5, and that version 3 is installed in the initial configuration. Then, for request `upgrade: gasoline-engine`, the solver generates:

- a constraint that prevents version 1 and 2 to be installed:

  ```
  gasoline-engine_1 + gasoline_engine_2 = 0
  ```

- a constraint to ensure the uniqueness of the installed version:

  ```
  gasoline-engine_3 + gasoline_engine_4  + gasoline_engine_5 = 1
  ```

4.1.2 Solving multicriteria optimisation

CUDFsolver implements a lexicographic order of multicriteria optimisation. Two different manners of combining different criteria have been considered: a classical lexicographic optimisation of each criteria in a sequential order and a mono criteria optimisation based on an aggregate of the multiple criteria. To illustrate these two methods, let us consider the two following criteria:

- Criterion (1) : minimize the number of removed functionalities among the installed ones. In other words, we should try to keep installed package p if any version of p is installed. This criterion requires the introduction of an additional binary variable p for each package. Remember that the default variables represent the status of a couple $< package, version >$, e.g., p_v which represents the status of package p version v. To

make sure that p is true if any version of p is installed, and that p is false otherwise, we add the following constraints:

$$-p + \sum_{v_i \in \mathcal{V}ersion(p)} p\_v_i \geq 0$$

and

$$n'' * p - \sum_{v_i \in \mathcal{V}ersion(p)} p\_v_i \geq 0$$

where $\mathcal{V}ersion(p)$ is the set of versions of p, and n'' is $\mathcal{C}ard(\mathcal{V}ersion(p))$, the cardinality of $\mathcal{V}ersion(p)$. Criterion(1) is then implemented by:

$$min \sum_{p \in F_{\mathcal{I}nstalled}} -p$$

where $F_{\mathcal{I}nstalled}$ is the set of installed functionalities.

- Criterion (2) : minimize the number of modifications, i.e. if package p_i, version v_i is installed keep it installed, if package p_u version v_u is uninstalled keep it uninstalled. Criterion (2) is implemented by:

$$min \sum_{p_i\_v_i \in P_{\mathcal{I}nstalled}} -p_i\_v_i + \sum_{p_u\_v_u \in P_{\mathcal{U}ninstalled}} p_u\_v_u$$

where $P_{\mathcal{I}nstalled}$ is the set of installed couples $< package, version >$ and $P_{\mathcal{U}ninstalled}$ is the set of uninstalled couples $< package, version >$.

These two criteria are considered in a lexical order so that they have to be handled independently.

Classical lexicographic multicriteria optimisation

Since the considered solvers optimize only one function at once, each criterion have to be handled one after the other. The first criterion is thus optimised according to the CUDF constraints:

$$\begin{aligned} & min \sum_{p \in F_{\mathcal{I}nstalled}} -p \\ & subject\ to \\ & \quad \mathcal{C}onst \end{aligned}$$

The underlying optimiser computes the global optima $c1$ of criterion $C1$ without taking into account the other criteria. The second criterion is then optimised with an additional constraint which states that first criterion value has to be less or equal to $c1$:

$$\begin{aligned} & min \sum_{p_i\_v_i \in P_{\mathcal{I}nstalled}} -p_i\_v_i + \sum_{p_u\_v_u \in P_{\mathcal{U}ninstalled}} p_u\_v_u \\ & subject\ to \\ & \quad \sum_{p \in F_{\mathcal{I}nstalled}} -p \leq c_1 \\ & \quad \mathcal{C}onst \end{aligned}$$

When there is more criteria, the process goes on, taking each previous criteria value into account through additional constraints:

$$\begin{aligned} & min\ Crit_i \\ subject\ to & \\ & Crit_1 \leq c_1 \\ & \dots \\ & Crit_{i-1} \leq c_{i-1} \\ & \mathcal{C}onst \end{aligned}$$

That way, each criteria is optimised according to previous criteria optima and without taking into account next criteria. It thus result in a lexicographic order of the multicriteria optimisation.

Note that each step requires a call to the underlying solver. This procedure can thus be time consuming.

Aggregate of multicriteria

To avoid calling the solver once for each criteria, we can aggregated criteria (1) and (2) in the following way:

$$min \sum_{p \in F_{\mathcal{I}nstalled}} -\mathcal{C}ard(P) * p + \sum_{p_i \in P_{\mathcal{I}nstalled}} -p_i + \sum_{p_u \in P_{\mathcal{U}ninstalled}} p_u$$

where $P = P_{\mathcal{I}nstalled} \cup P_{\mathcal{U}ninstalled}$. Multiplying first criterion coefficients by $\mathcal{C}ard(P)$ lets any of them have a higher value than any combination of the second criterion. That way, the first criterion could reach its minimum without being influenced by the second criterion. Note, however, that the variables involved in the first criterion are connected to variables involved in the second criterion by constraints.

When n criterion have to be handled, the first criterion is then multiplied by $(\mathcal{C}ard(P))^{n-1}$ and, thus can quickly reach the solver limitations. Moreover, the domain of the objective function can be huge. Note that solver limitations are not only linked to coefficient domains but also to the type of the coefficients. For example, Cplex uses double to do its computations and thus might represent values ranging from $-1.79 * 10^{308}$ to $1.79 * 10^{308}$. However, the distance between the double 10^{16} and its predecessor is equal to 2. As a consequence, if the objective function value is 10^{16} for one solution and $10^{16} - 1$ for another solution, Cplex might not be able to distinguish the two solutions.

4.2 Solver architecture and implementation

CUDFsolver has been interfaced with different solvers ranging from MIP solvers to pseudo boolean solvers. It relies on the capability of these different solvers to minimize a linear objective function subject to a set of linear equations and inequalities where coefficients are restricted to integers and variable domains to $\{0, 1\}$.

The main components of the solvers are:

- a CUDF problem reader which is in charge of reading the CUDF problem file and build the required internal data representation of the CUDF problem (i.e. a list of CUDFVersioned-

Package to represent the (*package*, *version*) couples and a list of CUDFVirtualPackage to represent a list of virtual packages, also called features).

- a constraint generator which generates constraints corresponding to the CUDF problem according to the chosen constraint solver.
- a CUDF problem solver that calls an external solver to get an optimal solution and generates the corresponding CUDF solution file.

Smooth interfacing of external solvers is facilitated by means of a solver abstract class.

Note that CUDFsolver is written in C++ and relies on Flex and Bison for the CUDF reader.

4.2.1 CUDF reader

For practical reasons, CUDF reader has been written has an independent library called `libcudf`. This library provides all the tools required to read and write CUDF files and acts as an abstract layer which handles CUDF files. Though its CUDF file writing capabilities, its main feature is to read a CUDF problem.

`libcudf` is organized around 4 files available in `libsrcs` directory:

- `cudf.h` provides CUDF data structures and CUDF function templates.
- `cudf.l` is a flex compliant lexical analyser for CUDF problems.
- `cudf.y` is bison compliant parser for CUDF problems which build the internal representation of a CUDF problem according to a CUDF problem file.
- `cudf_tools.c` implements methods to handle CUDF data structures, as well as, additional methods to print out a CUDF problem or a CUDF solution.

`libcudf` can be used as an independent library by any other solver.

4.2.2 constraint generation

`constraint_generation.c` (from `sources` directory) translates a CUDF problem into a minimization problem according to a chosen solver. To facilitate the interfacing with different solvers, class `abstract_solver` abstracts the solver interface. Each interface to a solver is then implemented through a concrete class (e.g., `cplex_solver.h` and `cplex_solver.c` for the cplex MILP solver).

4.2.3 CUDF problem solving

`cudf.c` (from `sources` directory) contains the main function. First, it calls the CUDF reader followed by a call to the constraint generator. Then, it calls the chosen solver (as an abstract solver) and produces the CUDF solution file thanks to the print out methods provided by `libcudf`.

5 INESC solver

INESC solver relies on Partial Weighted MaxSat (PWMS) to solve CUDF problems. On an implementation point of view, INESC solver calls sequentially the following components:

- `p2cudf` which translates the CUDF problem into a pseudo boolean optimisation problem. The translation done here follows similar principles than the one used in CUDF solver[1].
- `pb2wcnf` translates the pseudo boolean problem into a PWMS format.
- `MSUnCore`, a Partial Weighted MaxSat solver, is then used to solve the CUDF problem from the PWMS file.

This section focuses on the translation from the pseudo boolean problem to the partial weighted MaxSat problem and on solving of multicriteria optimisation problems using MSUnCore.

5.1 Multi-Level Optimization through PWMS

This section describes the use of the MSUnCore MaxSAT solver to get a solution from a CUDF instance. MSUnCore is a MaxSAT solver which is known for being particularly suitable for solving large problem instances coming from real applications. MSUnCore relies on the identification of unsatisfiable subsets of clauses. With this purpose, a SAT solver is called iteratively and relaxation variables are added to the clauses belonging to the unsatisfiable subset of clauses, jointly with at most one constraints for the new variables. At the end, the solution can be obtained from the values given to relaxation variables. Further details can be found in [MMSP09].

The core of the solver is called INESC-ID, as the solver has been developed within the participation of INESC-ID in the Mancoosi project[2]. Following the use of p2cudf.jar, we have to translate the OPB file produced by p2cudf.jar to the PWMS format. This translation is done with the tool pb2wcnf. Both tools, p2cudf.jar and pb2wcnf, also produce different mappings between the original source and the variables used in the output encoding. In the case of p2cudf.jar, it writes a file with the mapping between the variables of the OPB format and the package names and versions of the original CUDF file. The tool pb2wcnf produces the mapping between the Boolean variables and the OPB variables.

The way the different components of the solver have been integrated has several advantages. One of them is that each one of the main components of the process is a standalone tool that can be build independently from the others. Another advantage is that we can easily replace any of the components without affecting the other components. The main drawback of this method is that the communication between tools is currently achieved through files. It makes the implementation easier but can require significant memory and slows down the whole process if the files are too large.

5.1.1 Constraints and lexicographic optimization in MaxSAT

The goal of the tool pb2wcnf is to translate the pseudo-Boolean constraints and the objective function generated by p2cudf to a MaxSAT formula. Actually, the resulting formula corresponds

[1] p2cudf is more detailed in [ALL+10].
[2] Note, however, that the solver was also contributed by UdL and UCD, and uses p2cudf as a front-end.

to a Partial Weighted MaxSAT (PWMS) formula, where clauses can be hard or soft and soft clauses are associated with weights (being represented as (C, w)). A solution to a PWMS formula satisfies all the hard clauses and maximizes the sum of the weights associated to satisfied soft clauses. Note that in practice all the hard clauses are associated with a weight given by the sum of the weights of all soft clauses plus one.

The translation from PB-constraints to PWMS is performed as follows:

- The pseudo-Boolean constraints which correspond to clauses are translated to hard clauses, i.e., constraints corresponding to dependencies, conflicts, packages that cannot be found in the universe and packages that are requested to be installed, as well as, constraints corresponding to the handling of the inconsistency of installed packages.
- The remaining pseudo-Boolean constraints, which correspond to the cardinality constraint ≤ 1, meaning that no more than one version of each package can be installed, are converted to hard clauses as well. In this case, each constraint is encoded into a set of hard clauses using the bitwise encoding [Pre07]. Considering that we have m versions of the same package, the bitwise encoding introduces $O(log\ m)$ new variables and $O(m\ log\ m)$ binary clauses.

Moreover, the optimization function in the formula generated by p2cudf is translated to a set of weighted soft clauses having only one literal each. The weight associated to each clause is extracted from the coefficients in the optimization function. The translation is therefore performed as follows, depending on the criterion being used:

- In the paranoid criterion the optimization function is $(|U|+1)\times\sum removed_{p_i}+\sum changed_{p_j}$ where $|U|$ is U cardinality. Hence, for each package $removed_{p_i}$ is created a weighted soft clause $(\neg removed_{p_i}, |U|+1)$ and for each package $changed_{p_j}$ is created a weighted soft clause $(\neg changed_{p_j}, 1)$.
- In the trendy criterion the optimization function is $(|U|+1)\times(|U|+1)\times\sum removed_{p_i}+(|U|+1)\times\sum notuptodate_{p_j}+\sum new_{p_k}$. Hence, for each package $removed_{p_i}$ is created a weighted soft clause $(\neg removed_{p_i}, (|U|+1)\times(|U|+1))$, for each package $notuptodate_{p_j}$ is created a weighted soft clause $(\neg notuptodate_{p_j}, |U|+1)$ and for each package new_{p_k} is created a weighted soft clause $(\neg new_{p_k}, 1)$.

The weight of the hard clauses is therefore $(|U|+1)\times(|U|+1)$ for the paranoid criterion and $(|U|+1)\times(|U|+1)\times(|U|+1)$ for the trendy criterion.

5.1.2 Lexicographic Optimization with MaxSAT solving

Once we have translated the original CUDF problem instance to the OPB format, using p2cudf.jar, and the resulting OPB file to PWMS, using pb2wcnf, we can run the MaxSAT solver MSUnCore with the PWMS obtained file.

The actual version of MSUnCore being used has been enhanced with the possibility of handling lexicographic optimization functions as a sequence of optimization functions. This means that the optimal value for the first optimization criterion is found first, then the optimal value to the second criterion subject to the optimal value to the first criterion, and so on. (The use of this

approach both in the context of MaxSAT and PBO is further detailed in [ALMS09].) Not only this approach can be more efficient for solving some problems, but also it takes advantage of the competition evaluation rules which consider that better non-optimal solutions give better values to the most important criteria.

Let us illustrate a call to MSUnCore for which the optimization function corresponds to the trendy criterion. The soft clauses can therefore be of three different types: $(\neg removed_{p_i}, (|U|+1) \times (|U|+1))$, $(\neg notuptodate_{p_j}, |U|+1)$ and $(\neg new_{p_k}, 1)$, which requires three iterations:

1. In the first iteration, MSUnCore finds a solution to the hard clauses and the soft clauses with the largest weight (i.e. $(|U|+1) \times (|U|+1)$). Since all soft clauses have the same weight, the problem being solved is an instance of partial MaxSAT. The MinUNSAT solution computed by MSUnCore is denoted u_1. In addition, by the time the search has finished, MSUnCore has changed the original formula, by adding relaxation variables and at most one constraints as required to solve the problem while identifying unsatisfiable subsets of clauses. All clauses in the modified CNF formula are declared *hard* for the next iteration.

2. In the second iteration, MSUnCore finds a solution to the modified set of hard clauses and the soft clauses with the next weight (i.e. $(|U|+1)$). As before, all soft clauses have the same weight and so the problem being solved is an instance of partial MaxSAT. The new solution computed by MSUnCore is denoted u_2. In addition, and as before, all clauses in the modified CNF formula are declared *hard* for the next iteration.

3. Finally, for the third iteration, MSUnCore finds a solution to the modified set of hard clauses and the soft clauses with weight 1. Again MSUnCore solves an instance of partial MaxSAT. The new solution computed by MSUnCore is denoted u_3.

The final solution is given by u_1, u_2 and u_3. This approach has the advantage of completely eliminating the direct manipulation of weights by the MaxSAT solver. In addition, even when the solver has not yet run all the iterations, it is possible to provide a solution which can be already optimized for the first requirements. A more detailed description of this approach is described elsewhere [MSAGL10].

If MSUnCore ends with a solution within the available CPU time, we can proceed to parse the output of the solver and identify the Boolean variables set to `true`. Otherwise, if the solver cannot find a solution within the CPU time given, the solver outputs the best solution found so far and we can proceed the same way. A Boolean variable with the `true` value assigned means that the package referenced by this variable must be installed to get the optimum solution given by MSUnCore. Following the mapping from Boolean variables to OPB variables, and the mapping from OPB variables to package names, the package names and respective versions that have to be installed in the optimal solution can be identified. As a final step, the list of package names with the respective version is output as the CUDF solution.

6 An internal competition

At the beginning of January, for the Mancoosi first 2010 meeting, project members from University Paris Diderot organized an internal competition. This was a great opportunity to assess the solvers on a significant number of issues.

The solvers have been assessed on a set of 200 problems, about half being real problems and half being randomly generated problems. Note that more than 40 problems were coming from Caixa Magica, a Linux distributor.

Two tracks were organised corresponding to two different criteria. The first criteria, called "paranoid", proposed a conservative approach to the problem. In other words, the least changes would be the best. The second criteria attempted to provide a "fresher" installation and was called "trendy".

More formally, let U be the universe of a CUDF problem, containing the description of all available packages and, S a solution to the user request. Assume that $V(X, name)$ is the set of versions in which name (the name of a package) is installed in X, where X may be either U or S. Note that this set may be empty ($name$ is not installed), contains one element ($name$ is installed with exactly one version), or even contains multiple elements (if a package is installed in multiple versions). Finally, assuming that $H(name)$ is the highest version of name available in U and that $\mathcal{C}ard(X)$ is the cardinality of set X. Let us define some basic functions:

- $\#removed(U, S)$: is the number of packages $r$emoved in the proposed solution S w.r.t. the original installation U :

$$\#removed(U, S) = \mathcal{C}ard(\{name | V(U, name) \neq \emptyset \wedge V(S, name) = \emptyset\})$$

- $\#changed(U, S)$: is the number of packages with a $m$odified (set of) version(s) in the proposed solution S w.r.t. the original installation U :

$$\#changed(U, S) = \mathcal{C}ard(\{name | V(U, name) \neq V(S, name)\})$$

- $\#notuptodate(U, S)$: is the number of installed packages but $n$ot in the latest available version :

$$\#notuptodate(U, S) = \mathcal{C}ard(\{name | V(S, name) \neq \emptyset \wedge H(name) \notin V(S, name)\})$$

- $\#new(U, S)$: is the number of $n$ew packages in the proposed solution S w.r.t. the original installation U :

$$\#new(U, S) = \mathcal{C}ard(\{name | V(U, name) = \emptyset \wedge V(S, name) \neq \emptyset\})$$

The two optimization criteria combines basic criteria in a lexicographic order:

- paranoid: minimize the number of packages removed in the solution, and also the packages changed by the solution; the optimization criterion is :

$$lexicographic(min\ \ \#removed, min\ \ \#changed)$$

- trendy: minimize the number of packages removed in the solution, the number of outdated packages in the solution, and the number of extra packages installed. The optimization criterion is :

$$lexicographic(min\ \#removed, min\ \#notuptodate, min\ \#new)$$

Solvers taking part to the competition have been ranked according to these criteria.

Competition results[3] have underlined different solver issues. Most of the problems were linked to CUDF and criteria interpretation. Some of these issues have already been corrected and a new run of solvers have offered better results[4].

Note that, unfortunately, the solver from UCL was not ready in time to take part to the competition.

This competition was a good opportunity to assess most of the Mancoosi plugins and to see where and how these plugins could be improved.

7 A note on deliverable 4.2

Deliverable D4.2 hold the INESC-ID solver as well as the UNSA solver. Both solvers have been provided with their code and with a compiled version (for linux 32 bits).

INESC solver requires Java and Python and to run. This SAT-based solver uses the p2cudf parser and the MaxSAT solver MSUnCore. It should run smoothly on any linux 32 bits version.

Due to licence restrictions, the compiled UNSA solver provided here only uses free softwares. It includes the GNU glpk and lpsolve which are respectively distributed under GPL and LPGL. Even if cplex is not included, the sources contains the cplex interface and, thus, allow a compilation with cplex[5] Note that, though not included, it can also run using the pseudo boolean solver Wbo and the MILP solver SCIP.

Both solvers include a readme with additional information.

References

[ALL+10] Josep Argelich, Daniel Le Berre, Inês Lynce, Joao Marques-Silva, and Pascal Rapicault. Solving linux upgradeability problems using boolean optimization. In *Proceedings of LoCoCo2010 - Workshop on Logics for Component Configuration*, 2010.

[ALMS09] Josep Argelich, Inês Lynce, and Joao Marques-Silva. On solving Boolean multilevel optimization problems. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 393–398, July 2009.

[MMSP09] Vasco Manquinho, Joao Marques-Silva, and Jordi Planes. Algorithms for weighted Boolean optimization. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, pages 495–508, July 2009.

[3] See http://www.mancoosi.org/misc-nice-2010.
[4] See http://www.mancoosi.org/misc-internal/.
[5] Provided cplex is available...

[MSAGL10] Joao Marques-Silva, Josep Argelich, Ana Graça, and Inês Lynce. Boolean lexicographic optimization. In *Proceedings of the 17th RCRA International Workshop on "Experimental evaluation of algorithms for solving problems with combinatorial explosion"*, June 2010.

[Pre07] Steven David Prestwich. Variable dependency in local search: Prevention is better than cure. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, pages 107–120, 2007.

[TZ09] Ralf Treinen and Stefano Zacchiroli. Common upgradeability description format (cudf) 2.0. Technical report, MANCOOSI, November 2009.