Industrial-Size Scheduling with ASP+CP

Marcello Balduccini

Kodak Research Laboratories
Eastman Kodak Company
Rochester, NY 14650-2102 USA
marcello.balduccini@gmail.com

Abstract Answer Set Programming (ASP) combines a powerful, theoretically principled knowledge representation formalism and powerful solvers. To improve efficiency of computation on certain classes of problems, researchers have recently developed hybrid languages and solvers, combining ASP with language constructs and solving techniques from Constraint Programming (CP). The resulting ASP+CP solvers exhibit remarkable performance on "toy" problems. To the best of our knowledge, however, no hybrid ASP+CP language and solver have been used in practical, industrial-size applications. In this paper, we report on the first such successful application, consisting of the use of the hybrid ASP+CP system EZCSP to solve sophisticated industrial-size scheduling problems.

1 Introduction

Answer Set Programming (ASP) [9,11] combines a powerful, theoretically principled knowledge representation formalism and efficient computational tools called solvers. In the ASP programming paradigm, a problem is solved by writing an ASP program that defines the problem and its solutions so that the program's models (or, more precisely, answer sets) encode the desired solutions. State-of-the-art ASP solvers usually allow to compute the program's models quickly. The paradigm makes it thus possible not only to use the language to study sophisticated problems in knowledge representation and reasoning, but also to quickly write prototypes, and even evolve them into full-fledged applications.

The growth in the number of practical applications of ASP in recent years has highlighted, and allowed researchers to study and overcome, the limitations of the then-available solvers. Especially remarkable improvements have been brought about by the cross-fertilization with other model-based paradigms. This in fact resulted in the integration in ASP solvers of efficient computation techniques from those paradigms. In CLASP [7], for example, substantial performance improvements have been achieved by exploiting clause learning and backjumping. Some researchers have also advocated the use of specialized solving techniques for different parts of the program. In fact, most ASP solvers operate on *propositional* programs – and are sensitive to the size of such programs – which may make computations inefficient when the domains of some predicates' arguments are large. This is particularly the case of programs with predicates whose arguments range over subsets of $\mathcal{N}$, $\mathcal{Q}$, or $\mathcal{R}$. On the other hand, this is a common

situation in Constraint Programming (CP), and efficient techniques are available. For this reason, [3] proposed an approach in which ASP is extended with the ability to encode CP-style constraints, and the corresponding solver uses specialized techniques borrowed from CP solvers in handling such rules.

The research on integrating CP techniques in ASP has resulted in the development of various approaches, differing in the way CP constraints are represented in the hybrid language and in the way computations are carried out.

In [12] and [8], the language has been extended along the lines of [3], and specific ASP and CP solvers have been modified and integrated. In the former, syntax and semantics of ASP have been extended in order to allow representing, and reasoning about, quantities from $\mathcal{N}$, $\mathcal{Q}$, and $\mathcal{R}$. Separate solvers have been implemented for the different domains. In the latter, the focus has at least up to now been restricted to $\mathcal{N}$. The corresponding solver involves a remarkable integration of CP-solving techniques with clause learning and backjumping.

In [1], on the other hand, CP-style constraints have been encoded directly in ASP, without the need for extensions to the language. The corresponding representation technique allows dealing with quantities from $\mathcal{N}$, $\mathcal{Q}$, and $\mathcal{R}$, while the EZCSP solver allows selecting the most suitable ASP and CP solvers without the need for modifications. The current implementation of the solver supports $\mathcal{N}$ (more precisely, finite domains), $\mathcal{Q}$, and $\mathcal{R}$.

Although the above hybrid ASP+CP systems have shown remarkable performance on "toy" problems, to the best of our knowledge none of them has yet been confronted with practical, industrial-size applications. In this paper, we report on one such successful application, consisting in the use of EZCSP to solve sophisticated industrial-size scheduling problems. We hope that, throughout the paper, the reader will be able to appreciate how elegant, powerful, and elaboration tolerant the EZCSP formalization is.

The application domain of interest in this paper is that of industrial printing. In a typical scenario for this domain, orders for the printing of books or magazines are more or less continuously received by the print shop. Each order involves the execution of multiple jobs. First, the pages are *printed* on (possibly different) press sheets. The press sheets are often large enough to accommodate several (10 to 100) pages, and thus a suitable layout of the pages on the sheets must be found. Next, the press sheets are *cut* in smaller parts called *signatures*. The signatures are then *folded* into booklets whose page size equals the intended page size of the order. Finally the booklets are *bound* together to form the book or magazine to be produced. The decision process is made more complex by the fact that multiple models of devices may be capable of performing a job. Furthermore, many decisions have ramifications and inter-dependencies. For example, selecting a large press sheet would prevent the use of a small press. The underlying decision-making process is often called *production planning* (the term "planning" here is only loosely related to the meaning of planning the execution of actions over time typical in the ASP community, but is retained because it is relatively well established in the field of the application). Another set of decisions deals with *scheduling*. Here one needs to determine *when* the various jobs will

be executed using the devices available in the print shop. Multiple devices of the same model may be available, thus even competing jobs may be run in parallel. Conversely, some of the devices can be offline – or go suddenly offline while production is in progress – and the scheduler must work around that. Typically, one wants to find a schedule that minimizes the tardiness of the orders while giving priority to the more important orders. Since orders are received on a continuous basis, one needs to be able to update the schedule in an incremental fashion, in a way that causes minimal disruption to the production, and can satisfy *rush orders*, which need to be executed quickly and take precedence over the others. Similarly, the scheduler needs to react to sudden changes in the print shop, such as a device going offline during production.

In this paper we describe the use of ASP+CP for the scheduling component of the system. It should be noted that the "toy" problems on which ASP+CP hybrids have been tested so far also include scheduling domains (see e.g. the 2nd ASP Competition [6]). However, as we hope will become evident later on, the constraints imposed on our system by practical use make the problem and the solution substantially more sophisticated, and the encoding and development significantly more challenging.

We begin by providing background on EZCSP. Next, we provide a general mathematical definition of the problem for our application domain. Later, we encode a subclass of problems of interest in EZCSP and show how schedules can be found by computing the extended answer sets of the corresponding encodings.

2 Background

In this section we provide basic background on EZCSP. The interested reader can find more details in [1]. Atoms and literals are formed as usual in ASP. A *rule* is a statement of the form $h \leftarrow l_1, \ldots, l_m, \text{not } l_{m+1}, \ldots, \text{not } l_n$, where h and l_i's are literals and *not* is the so-called *default negation*. The intuitive meaning of the rule is that a reasoner who believes $\{l_1, \ldots, l_m\}$ and has no reason to believe $\{l_{m+1}, \ldots, l_n\}$, has to believe h. A *program* is a set of rules. A rule containing variables is viewed as short-hand for the set of rules, called *ground instances*, obtained by replacing the variables by all possible ground terms. The ground instance of a program is the collection of the ground instances of its rules. Because of space considerations, we simply define an answer set of a program Π as one of the sets of brave conclusions entailed by Π under the answer set semantics. The precise definition can be found in [9]. Throughout this paper, readers who are not familiar with the definition can rely on the intuitive reading of ASP rules given above.

The definition of constraint satisfaction problem that follows is adapted from [13]. A *Constraint Satisfaction Problem (CSP)* is a triple $\langle X, D, C \rangle$, where $X = \{x_1, \ldots, x_n\}$ is a set of variables, $D = \{D_1, \ldots, D_n\}$ is a set of domains, such that D_i is the domain of variable x_i (i.e. the set of possible values that the variable can be assigned), and C is a set of constraints. Each constraint $c \in C$ is a pair $c = \langle \sigma, \rho \rangle$ where σ is a list of variables and ρ is a subset of the

Cartesian product of the domains of such variables. An *assignment* is a pair $\langle x_i, a\rangle$, where $a \in D_i$, whose intuitive meaning is that variable x_i is assigned value a. A *compound assignment* is a set of assignments to distinct variables from X. A *complete assignment* is a compound assignment to all of the variables in X. A constraint $\langle \sigma, \rho\rangle$ specifies the acceptable assignments for the variables from σ. We say that such assignments *satisfy* the constraint. A *solution* to a CSP $\langle X, D, C\rangle$ is a complete assignment that satisfies every constraint from C. Constraints can be represented either *extensionally*, by specifying the pair $\langle \sigma, \rho\rangle$, or *intensionally*, by specifying an expression involving variables, such as $x < y$. In this paper we focus on constraints represented intensionally. A *global constraint* is a constraint that captures a relation between a non-fixed number of variables [10], such as $sum(x, y, z) < w$ and $all\_different(x_1, \ldots, x_k)$. One should notice that the mapping of an intensional constraint specification into a pair $\langle \sigma, \rho\rangle$ depends on the *constraint domain*. For this reason, in this paper we assume that every CSP includes the specification of the intended constraint domain.

In EZCSP, programs are written in such a way that their answer sets encode the desired CSPs. The solutions to the CSPs are then computed using a CP solver. CSPs are encoded in EZCSP using the following three types of statements: (1) a constraint domain declaration, i.e. a statement of the form $cspdomain(\mathcal{D})$, where $\mathcal{D}$ is a constraint domain such as fd, q, or r; informally, the statement says that the CSP is over the specified constraint domain (finite domains, $\mathcal{Q}$, $\mathcal{R}$), thereby fixing an interpretation for the intensionally specified constraints; (2) a constraint variable declaration, i.e. a statement of the form $cspvar(x, l, u)$, where x is a ground term denoting a variable of the CSP (CSP variable or constraint variable for short), and l and u are numbers from the constraint domain; the statement says that the domain of x is $[l, u]$;(3) a constraint statement, i.e. a statement of the form $required(\gamma)$, where γ is an expression that intensionally represents a constraint on (some of) the variables specified by the *cspvar* statements; intuitively the statement says that the constraint intensionally represented by γ is required to be satisfied by any solution to the CSP. For the purpose of specifying global constraints, we allow γ to contain expressions of the form $[\delta/k]$. If δ is a function symbol, the expression intuitively denotes the sequence of all variables formed from function symbol δ and with arity k, ordered lexicographically. If δ is a relation symbol, the expression intuitively denotes the sequence $\langle e_1, e_2, \ldots, e_n\rangle$ where e_i is the last element of the i^{th} k-tuple satisfying relation δ, according to the lexicographic ordering of such tuples.

Example 1. We are given 3 variables, v_1, v_2, v_3, ranging over $[1, 5]$ and we need to find values for them so that $v_2 - v_3 > 1$ and their sum is greater than or equal to 4. A possible encoding of the problem is $A_1 = \{cspdomain(fd), cspvar(v(1), 1, 5), cspvar(v(2), 1, 5), cspvar(v(3), 1, 5), required(v(2) - v(3) > 1), required(sum([v/1]) \geq 4)\}$ ◇

Let A be a set of atoms, including atoms formed from relations *cspdomain*, *cspvar*, and *required*. We say that A is a *well-formed CSP definition* if: (1) A contains exactly one constraint domain declaration; (2) the same CSP variable does not occur in two or more constraint variable declarations of A; and (3)

every CSP variable that occurs in a constraint statement from A also occurs in a constraint variable declaration from A.

Let A be a well-formed CSP definition. The CSP *defined* by A is the triple $\langle X, D, C\rangle$ such that: (1) $X = \{x_1, x_2, \ldots, x_k\}$ is the set of all CSP variables from the constraint variable declarations in A; (2) $D = \{D_1, D_2, \ldots, D_k\}$ is the set of domains of the variables from X, where the domain D_i of variable x_i is given by arguments l and u of the constraint variable declaration of x_i in A, and consists of the segment between l and u in the constraint domain specified by the constraint domain declaration from A; (3) C is a set containing a constraint γ' for each constraint statement $required(\gamma)$ of A, where γ' is obtained by: (a) replacing the expressions of the form $[f/k]$, where f is a function symbol, by the list of variables from X formed by f and of arity k, ordered lexicographically; (b) replacing the expressions of the form $[r/k]$, where r is a relation symbol, by the sequence $\langle e_1, \ldots, e_n\rangle$, where, for each i, $r(t_1, t_2, \ldots, t_{k-1}, e_i)$ is the i^{th} element of the sequence, ordered lexicographically, of atoms from A formed by relation r; (c) interpreting the resulting intensionally specified constraint with respect to the constraint domain specified by the constraint domain declaration from A. A pair $\langle A, \alpha\rangle$ is an *extended answer set* of program Π iff A is an answer set of Π and α is a solution to the CSP defined by A.

Example 2. Set A_1 from Example 1 defines the CSP:

$$\langle \{v_1, v_2, v_3\}, \left\{ \begin{array}{c} \{1,2,3,4,5\}, \{1,2,3,4,5\}, \\ \{1,2,3,4,5\} \end{array} \right\}, \left\{ \begin{array}{c} v_2 - v_3 > 1, \\ sum(v(1), v(2), v(3)) \geq 4 \end{array} \right\} \rangle.$$

3 Problem Definition

One distinguishing feature of ASP and derived languages is that they allow one to encode a problem at a level of abstraction that is close to that of the problem's mathematical (or otherwise precisely formulated, see e.g. [2]) specification. Consequently, it is possible for the programmer (1) to inspect specification and encoding and convince himself of the correctness of the encoding, and (2) to accurately prove the correctness of the encoding with respect to the formalization with more ease than using other approaches. *The ability to do this is quite important in industrial applications, where one assumes responsibility towards the customers for the behavior of the application.* Therefore, in this paper we precede the description of the encoding with a precisely formulated problem definition. Let us begin by introducing some terminology.

By *device class* (or simply *device*) we mean a type or model of artifact capable of performing some phase of production, e.g. a press model XYZ or a folder model MNO. By *device-set* of a print shop we mean the set of devices available in the shop. By *device instance* (or simply *instance*) we mean a particular exemplar of a device class. For example, a shop may have multiple XYZ presses, each of which is a device instance. We denote the fact that instance i is an instance of device class d by the expression $i \in d$.

A *job* j is a pair $\langle len, devices\rangle$ where len is the length of the job, and $devices \subseteq$ *device-set* is a set of devices that are capable of performing the job.

Given a job j, the two components are denoted by $len(j)$ and $devices(j)$. Intuitively, the execution of a job can be split among a subset of the instances of the elements of $devices(j)$. A *job-set* J is a pair $\langle \Gamma, PREC \rangle$, where Γ is a set of jobs, and $PREC$ is a directed acyclic graph over the elements of Γ, intuitively describing a collection of precedences between jobs. An arc $\langle j_1, j_2 \rangle$ in $PREC$ means that the execution of job j_1 must precede that of job j_2 (that is, the execution of j_1 must be completed before j_2 can be started). The components of J are denoted, respectively, by Γ_J and $PREC_J$.

The *usage-span* of an instance i is a pair $\langle start\text{-}time_i, duration_i \rangle$, intuitively stating that instance i will be in use (for a given purpose) from $start\text{-}time_i$ and for the specified duration. By US we denote the set of all possible usage-spans. Given a usage-span u, $start(u)$ denotes its first component and $dur(u)$ denotes its second. In the remainder of the discussion, given a partial function f, we use the notation $f(\cdot) = \perp$ (resp., $f(\cdot) \neq \perp$) to indicate that f is undefined (resp., defined) for a given tuple.

Definition 1 (Work Assignment). *A* work assignment *for a job-set J is a pair $\langle I, U \rangle$, where: (1) I is a function that associates to every job $j \in \Gamma_J$ a set of device instances (we use I_j as an alternative notation for $I(j)$), (2) U is a collection of (partial) functions* $\text{usage}_\text{j} : I_j \rightarrow US$ *for every $j \in \Gamma_J$, and the following requirements are satisfied:*

1. *(I_j is valid) For every $i \in I_j$, $\exists d \in devices(j)$ s.t. $i \in d$.*
2. *($usage_j$ is valid) $\forall usage_j \in U$, $\sum_{i \in I_j, usage_j(i) \neq \perp} dur(usage_j(i)) = len(j)$.*
3. *(overlap) For any two jobs $j \neq j'$ from Γ_J and every $i \in I_j \cap I_{j'}$ such that $usage_j(i) \neq \perp$ and $usage_{j'}(i) \neq \perp$:*

$$start(usage_j(i)) + dur(usage_j(i)) \leq start(usage_{j'}(i)), \quad or$$
$$start(usage_{j'}(i)) + dur(usage_{j'}(i)) \leq start(usage_j(i)).$$

4. *(order) For every $\langle j, j' \rangle \in PREC_J$, j' starts only after j has been completed.*

$\diamond$

A *single-instance work assignment* for job-set J is a work assignment $\langle I, U \rangle$ such that $\forall j \in \Gamma_J, |I_j| = 1$. A single-instance work assignment intuitively prescribes the use of exactly one device instance for each job. In the rest of the discussion, we will focus mainly on a special type of job-set: a *simplified job-set* is a job-set J such that, for every job j from Γ_J, $|devices(j)| = 1$. Notice that work assignments for a simplified job-set are not necessarily single-instance. If multiple instances of some device are available, it is still possible to split the work for a job between the instances, or assign it to a particular instance depending on the situation. Next, we define various types of scheduling problems of interest and their solutions.

A *deadline-based decision (scheduling) problem* is a pair $\langle J, d \rangle$ where J is a set of jobs and d is a (partial) *deadline function* $d : J \rightarrow \mathcal{N}$, intuitively specifying deadlines for the jobs, satisfying the condition:

$$\forall j, j' \in \Gamma_J \quad [\ \langle j, j' \rangle \in PREC_J \rightarrow d(j) = \perp\ ]. \tag{1}$$

A *solution to a deadline-based decision problem* $\langle J, d\rangle$ is a work assignment $\langle I, U\rangle$ such that, for every $j \in \Gamma_J$: $\forall i \in I_j$ [$d(j) \neq \perp \rightarrow start(usage_j(i)) + dur(usage_j(i)) \leq d(j)$].

A *cost-based decision problem* is a tuple $\langle J, c, k\rangle$ where J is a job-set, c is a *cost function* that associates a cost (represented by a natural number) to every possible work assignment of J, and k is a natural number, intuitively corresponding to the target cost. A *solution to a cost-based decision problem* $\langle J, c, k\rangle$ is a work assignment W such that

$$c(W) \leq k. \tag{2}$$

An *optimization problem* is a pair $\langle J, c\rangle$ where J is a job-set and c is a *cost function* that associates a cost to every possible work assignment of J. A *solution to an optimization problem* $P = \langle J, c\rangle$ is a work assignment W such that, for every other work assignment W', $c(W) \leq c(W')$.

Since a solution to an optimization problem $\langle J, c\rangle$ can be found by solving the sequence of cost-based decision problems $\langle J, c, 0\rangle, \langle J, c, 1\rangle, \ldots$, in the rest of this paper we focus on solving decision problems. Details on how the optimization problem is solved by our system will be discussed in a longer paper.

In our system, scheduling can be based on total tardiness, i.e. on the sum of the amount of time by which the jobs are past their deadline. Next, we show how this type of scheduling is an instance of a cost-based decision problem.

Example 3. Total tardiness
We are given a job-set J, the target total tardiness k, and a deadline function d (as defined earlier). We construct a cost function c so that the value of $c(W)$ is the total tardiness of work assignment W. The construction is as follows. First we define auxiliary function $c'(j, W)$ which computes the tardiness of job $j \in \Gamma_J$ based on W:

$$c'(j, W) = \begin{cases} 0 & \text{if } d(j) = \perp \text{ or } usage_j(i) = \perp \\ \max(0, start(usage_j(i)) + dur(usage_j(i)) - d(j)) & \text{otherwise.} \end{cases}$$

Now we construct the cost function as: $c(W) = \sum_{j \in \Gamma_J} c'(j, W)$. The work assignments with total tardiness less than or equal to some value k are thus the solutions of the cost-based decision problem $\langle J, c, k\rangle$. ◇

At this stage of the project,we focus on simplified job-sets and single-instance solutions. Furthermore, all instances of a given device are considered to be identical (one could deal with differences due e.g. to aging of some instances by defining different devices).

Under these conditions, a few simplifications can be made. Because we are focusing on single-instance solutions, given a work assignment $\langle I, U\rangle$, it is easy to see that I_j is a singleton for every j. Moreover, since we are also focusing on simplified job-sets, for every j, $usage_j$ is defined for a single device instance of a single device $d \in devices(j)$. It is not difficult to see that, for the usage-span $\langle start\text{-}time, duration\rangle$ of every job j, $duration = len(j)$. Thus, the only information that needs to be specified for work assignments is the start time of each job j and the device instance used for the job.

A solution to a scheduling problem can now be more compactly described as follows. A *device schedule* for d is a pair $\langle L, S\rangle$, where $L = \langle j_1, \ldots, j_k\rangle$ is a sequence of jobs, and $S = \langle s_1, \ldots, s_k\rangle$ is a sequence of integers. Intuitively, L is the list of jobs that are to be run on device d, and S is a list of start times such that each s_m is the start time of job j_m. A *global schedule* for a set of devices D is a function σ that associates each device from D with a device schedule.

4 Encoding and Solving Scheduling Problems

In this section, we describe how scheduling problems for our application domain are encoded and solved using EZCSP. Although our formalization is largely independent of the particular constraint domain chosen, for simplicity we fix the constraint domain to be that of finite domains, and assume that any set of rules considered also contains the corresponding specification $cspdomain(fd)$.

A device d with n instances is encoded by the set of rules $\varepsilon(d) = \{device(d).\ \ instances(d, n).\}$. A job j is encoded by $\varepsilon(j) = \{job(j).\ \ job\_len(j, l).\ \ job\_device(j, d).\}$, where $l = len(j)$ and $d \in device(j)$ ($device(j)$ is a singleton under the conditions stated earlier).

The components of a job-set $J = \langle \Gamma, PREC\rangle$ are encoded as follows:

$$\varepsilon(\Gamma) = \bigcup_{j \in \Gamma_J} \varepsilon(j) \ \ ; \ \ \varepsilon(PREC_J) = \{precedes(j, j').\mid \langle j, j'\rangle \in PREC_J\}.$$

The encoding of job-set J is $\varepsilon(J) = \varepsilon(\Gamma_J) \cup \varepsilon(PREC_J)$.

Given a scheduling problem P, the overall goal is to specify its encoding $\varepsilon(P)$. In this section we focus on solving cost-based decision problems $\langle J, c, k\rangle$ with c based on total tardiness, and defined as in Example 3. Our goal then is to provide the encoding $\varepsilon(\langle J, c, k\rangle)$.

We represent the start time of the execution of job j on some instance of device d by constraint variable $st(d, j)$. The definition of the set of such constraint variables is given by ε_{vars}, consisting of the rule:

$$cspvar(st(D, J), 0, MT) \leftarrow job(J),\ job\_device(J, D),\ max\_time(MT).$$

together with the definition of relation max_time, which determines the upper bound of the range of the constraint variables. The next constraint ensures that the start times satisfy the precedences in $PREC_J$:

$$\varepsilon_{prec} = \begin{cases} required(st(D2, J2) \geq st(D1, J1) + Len1) \leftarrow \\ \qquad job(J1),\ job(J2),\ job\_device(J1, D1),\ job\_device(J2, D2), \\ \qquad precedes(J1, J2),\ job\_len(J1, Len1). \end{cases}$$

The deadlines of all jobs $j \in \Gamma_J$ are encoded by a set ε_{dl} of facts of the form $deadline(j, n)$. One can ensure that requirement (1) is satisfied by specifying a constraint $\leftarrow precedes(J1, J2),\ deadline(J1, D)$. Function c' from Example 3 is encoded by introducing an auxiliary constraint variable $td(j)$ for every job

j, where $td(j)$ represents the value of $c'(j, W)$ for the current work assignment. The encoding of c', $\varepsilon(c')$, consists of $\varepsilon_{var} \cup \varepsilon_{prec} \cup \varepsilon_{dl}$ together with:

$$cspvar(td(J), 0, MT) \leftarrow job(J),\ max\_time(MT).$$
$$required(td(J) == max(0, st(D, J) + Len - Deadline)) \leftarrow$$
$$job(J),\ job\_device(J, D),\ deadline(J, Deadline),\ job\_len(J, Len).$$

Notice that the constraint is only enforced if a deadline has been specified for job j. An interesting way to explicitly set the value of $td(j)$ to 0 for all other jobs consists in using:

$$enforced(td(J)) \leftarrow job(J),\ required(td(J) == X),\ X \neq 0.$$
$$required(td(J) == 0) \leftarrow job(J),\ \text{not}\ enforced(td(J)).$$

Function c from Example 3 is encoded by introducing an auxiliary constraint variable *tot_tard*. The encoding, $\varepsilon(c)$, consists of $\varepsilon(c')$ together with:

$$cspvar(tot\_tard, 0, MT) \leftarrow max\_time(MT).$$
$$required(sum([td/1], ==, tot\_tard)).$$

where the constraint intuitively computes the sum of all constraint variables $td(\cdot)$. The final step in encoding the decision problem $\langle J, c, k\rangle$ is to provide a representation, $\varepsilon(k)$, of constraint (2), which is accomplished by the rule:

$$required(tot\_tard \leq K) \leftarrow max\_total\_tardiness(K).$$

together with the definition of relation *max_total_tardiness*, specifying the maximum total tardiness allowed. It is interesting to note the flexibility of this representation: if relation *max_total_tardiness* is not defined, then the above constraint is not enforced – in line with the informal reading of the rule – and thus one can use the encoding to find a schedule irregardless of its total tardiness.

The encoding of a cost-based decision problem $\langle J, c, k\rangle$, where c computes total tardiness, is then: $\varepsilon(\langle J, c, k\rangle) = \varepsilon(J) \cup \varepsilon(c) \cup \varepsilon(k)$.

Now that we have a complete encoding of the problem, we discuss how scheduling problems are solved. Given $\varepsilon(\langle J, c, k\rangle)$, to solve the scheduling problem we need to assign values to the constraint variables while enforcing the following requirements:

[Overlap] if two jobs j and j', being executed on the same device, overlap (that is, one starts before the other is completed), then they must be executed on two separate instances of the device;

[Resources] at any time, no more instances of a device can be used than are available.

This can be accomplished compactly and efficiently using global constraint *cumulative* [4]. The *cumulative* constraint takes as arguments: (1) a list of constraint variables encoding start times; (2) a list specifying the execution length of each job whose starts time is to be assigned; (3) a list specifying the amount of resources required for each job whose start time is to be assigned; (4) the

maximum number of resources available at any time on the device. In order to use *cumulative*, we represent the number of available device instances as an amount of resources, and use a separate cumulative constraint for the schedule of each device. The list of start times for the jobs that are to be processed by device d can be specified, in EZCSP, by a term $[st(d)/2]$, intuitively denoting the list of terms (1) formed by function symbol st, (2) of arity 2, and (3) with d as first argument. We also introduce auxiliary relations len_by_dev and res_by_dev, which specify, respectively, the length and number of resources for the execution on device d of job j. The auxiliary relations are used to specify the remaining two lists for the *cumulative* constraint, using EZCSP terms $[len\_by\_dev(D)/3]$ and $[res\_by\_dev(D)/3]$. The complete scheduling module, Π_{solv}, is:

$$required(cumulative([st(D)/2], [len\_by\_dev(D)/3], [res\_by\_dev(D)/3], N)) \leftarrow instances(D, N).$$

$$len\_by\_dev(D, J, N) \leftarrow job(J),\ job\_device(J, D),\ job\_len(J, N).$$

$$res\_by\_dev(D, J, 1) \leftarrow job(J),\ job\_device(J, D).$$

Notice that, since we identify the resources with the instances of a device, the amount of resources required by any job j at any time is 1.

Schedules for a cost-based decision problem $\langle J, c, k\rangle$, where c computes total tardiness, can be then found by computing the extended answer sets of the program $\varepsilon(\langle J, c, k\rangle) \cup \Pi_{solv}$. Using the definitions from Section 3, one can prove the following:

Proposition 1. *Let $\langle J, c, k\rangle$ be a cost-based decision problem, where c computes total tardiness. The global schedules of $\langle J, c, k\rangle$ are in one-to-one correspondence with the extended answer sets of the program $\varepsilon(\langle J, c, k\rangle) \cup \Pi_{solv}$.*

5 Incremental and Penalty-Based Scheduling

In this section, we describe the solution to more sophisticated scheduling problems. To give more space to the encoding, we omit the precise problem definitions, which can be obtained by extending the definitions from Section 3.

It is important to stress that *these extensions to the scheduler are entirely incremental, with no modifications needed to the encoding from Section 4.* This remarkable and useful property is the direct result of the elaboration tolerance typical of ASP encodings.

The first extension of the scheduler consists in considering a more sophisticated cost-based decision problem, $\langle J, c^*, k^*\rangle$. In everyday use, some orders take precedence over others, depending on the service level agreement the shop has with its customers. Each service level intuitively yields a different penalty if the corresponding jobs are delivered late. The total penalty of a schedule is thus obtained as the weighted sum of the tardiness of the jobs, where the weights are based on each job's service level. The encoding, $\varepsilon(c^*)$, of c^* consists of $\varepsilon(c)$

together with (relation *weight* defines the assignments of weights to jobs):

$$cspvar(penalty(J), 0, MP) \leftarrow job(J),\ max\_penalty(MP).$$
$$required(penalty(J) == td(J, O) * Weight) \leftarrow job(J),\ weight(J, Weight).$$
$$cspvar(tot\_penalty, 0, MP) \leftarrow max\_penalty(MP).$$
$$required(sum([penalty/2], ==, tot\_penalty)).$$

The encoding of $\varepsilon(k^*)$ extends $\varepsilon(k)$ by the rule:

$$required(tot\_penalty \leq P) \leftarrow max\_total\_penalty(P).$$

which encodes constraint (2) for penalties. Notice that, thanks to the elaboration tolerance of the encoding developed in Section 4, it is safe for $\varepsilon(k^*)$ to include $\varepsilon(k)$, since now relation *max_total_tardiness* is left undefined.

Another typical occurrence in everyday use is that the schedule must be updated incrementally, either because new orders were received, or because of equipment failures. During updates, intuitively one needs to avoid re-scheduling jobs that are already being executed. We introduce a new decision problem $\langle J^*, c^*, k^* \rangle$, where c^*, k^* are as above, and J^* extends J to include information about the current schedule; $\varepsilon(J^*)$ includes $\varepsilon(J)$ and is discussed next.

To start, we encode the current schedule by relations $curr\_start(j, t)$ and $curr\_device(j, d)$. The current (wall-clock) time t is encoded by relation $curr\_time(t)$. The following rules informally state that, if according to the current schedule production of a job has already started, then its start time and device must remain the same. Conversely, all other jobs must have a start time that is no less than the current time.[1]

$$already\_started(J) \leftarrow curr\_start(J, T),\ curr\_time(CT),\ CT > T.$$
$$must\_not\_schedule(J) \leftarrow already\_started(J),\ \text{not}\ ab(must\_not\_schedule(J)).$$
$$required(st(D, J) \geq CT) \leftarrow job\_device(J, D),\ curr\_time(CT),\ \text{not}\ must\_not\_schedule(J).$$
$$required(st(D, J) == T) \leftarrow curr\_device(J, D),\ curr\_start(J, T),\ must\_not\_schedule(J).$$

It is worth noting that the use of a default to define *must_not_schedule* allows extending the scheduler in a simple and elegant way by defining exceptions.

Whereas the above rules allow scheduling new jobs without disrupting current production, the next set deals with equipment going offline, even during production. Notice that by "equipment" we mean a particular instance of a device. To properly react to this situation, the system first needs to have an explicit representation of which device instance is assigned to perform which job. This can be accomplished by introducing a new variable $on\_instance(j)$. The value of the variable is a number that represents the device instance assigned to the job (among the instances of the device prescribed by $job\_device(j, d)$). The

[1] One may not want to schedule jobs to start exactly at the current time, as there would not be time to move the supplies and set up the job. Extending the rules to accomplish that is straightforward.

corresponding variable declaration and constraints are encoded by:

$$cspvar(on\_instance(J), 1, N) \leftarrow job\_device(J, D),\ instances(D, N).$$

$$required((on\_instance(J1) \neq on\_instance(J2)) \ \vee \\ (st(D, J2) \geq st(D, J1) + Len1) \ \vee \ (st(D, J1) \geq st(D, J2) + Len2)) \leftarrow \\ job\_device(J1, D),\ job\_device(J2, D),\ J1 \neq J2, \\ len(J1, Len1),\ len(J2, Len2),\ instances(D, N),\ N > 1.$$

$$required(on\_instance(J) \neq I) \leftarrow \\ job\_device(J, D),\ offline\_instance(D, I),\ \text{not } must\_not\_schedule(J).$$

The second rule intuitively says that, if d has more than one instance and is scheduled to run two (or more) jobs, then either a job ends before the other starts, or the jobs must be assigned to two different instances. The rule is an example of use of *reified constraints*, as defined for example in [5]. The last rule says that no job can be assigned to an offline instance. For incremental scheduling, we also extend the encoding of the current schedule by introducing a relation $curr\_on\_instance(j, i)$, which records the instance-job previously determined.

The reader might wonder about the overlap between *cumulative*, used earlier, and the above constraints. In fact, *cumulative* already performs a limited form of reasoning about instance-job assignments when it determines if sufficient resources are available to perform the jobs. Although it is possible to replace *cumulative* by a set of specific constraints for the assignment of start times, we believe that using *cumulative* makes the encoding more compact and readable.

The encoding $\varepsilon(J^*)$ is completed by rules that detect situations in which a device went offline while executing a job. Relation $offline\_instance(d, i)$ states that instance i of device d is currently offline.

$$already\_finished(J) \leftarrow \ curr\_start(J, T),\ len(J, Len),\ curr\_time(CT),\ CT \geq T + Len.$$

$$ab(must\_not\_schedule(J)) \leftarrow \\ already\_started(J),\ \text{not } already\_finished(J),\ curr\_device(J, D), \\ curr\_on\_instance(J, I),\ offline\_instance(D, I).$$

The last rule defines an exception to the default introduced earlier. Informally, the rule says that if instance i is offline, any job assigned to it that is currently in production constitutes an exception to the default. It is not difficult to see that such exceptional jobs are subjected to regular rescheduling. The presence of atoms formed by relation *ab* in the extended answer set is also used by the system to warn the user that a reschedule was forced by a device malfunction.

In conclusion, the solutions to problem $\langle J^*, c^*, k^* \rangle$ can be found by computing the extended answer sets of $\varepsilon(\langle J^*, c^*, k^* \rangle)$.

6 Conclusions

In this paper we have described what to the best of our knowledge is the first industrial-size application of an ASP+CP hybrid language. The application is currently being considered for use in commercial products. Performance evaluation of our system is under way. A simplified version of the domain has been

accepted as a benchmark for the Third ASP Competition at LPNMR-11. Preliminary analysis on customer data showed that performance is comparable to that of similar implementations written using CP alone, *with schedules for customer-provided instances typically found in less than one minute, and often in a few seconds*. In comparison with direct CP encodings, we found that the compactness, elegance, and elaboration tolerance of the EZCSP encoding are superior. In a CLP implementation that we have developed for comparison, the number of rules in the encoding was close to one order of magnitude larger than the number of rules in the EZCSP implementation. Moreover, writing those rules often required one to consider issues with procedural flavor, such as how and where certain information should be collected.

References

1. Balduccini, M.: Representing Constraint Satisfaction Problems in Answer Set Programming. In: ICLP09 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP09) (Jul 2009)
2. Balduccini, M., Girotto, S.: Formalization of Psychological Knowledge in Answer Set Programming and its Application. Journal of Theory and Practice of Logic Programming (TPLP) 10(4–6), 725–740 (Jul 2010)
3. Baselice, S., Bonatti, P.A., Gelfond, M.: Towards an Integration of Answer Set and Constraint Solving. In: Proceedings of ICLP 2005 (2005)
4. Beldiceanu, N., Contejean, E.: Introducing Global Constraints in CHIP. Mathl. Comput. Modelling 20(12), 97–123 (1994)
5. Carlson, B., Carlsson, M., Ottosson, G.: An Open-Ended Finite Domain Constraint Solver. In: Proc. Programming Languages: Implementations, Logics, and Programs (1997)
6. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczynski, M.: The Second Answer Set Programming Competition. In: 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR09). pp. 637–654 (Sep 2009)
7. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Veloso, M.M. (ed.) Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07). pp. 386–392. MIT Press (2007)
8. Gebser, M., Ostrowski, M., Schaub, T.: Constraint Answer Set Solving. In: 25th International Conference on Logic Programming (ICLP09). vol. 5649 (2009)
9. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9, 365–385 (1991)
10. Katriel, I., van Hoeve, W.J.: Handbook of Constraint Programming, chap. 6. Global Constraints, pp. 169–208. Foundations of Artificial Intelligence, Elsevier (2006)
11. Marek, V.W., Truszczynski, M.: The Logic Programming Paradigm: a 25-Year Perspective, chap. Stable models and an alternative logic programming paradigm, pp. 375–398. Springer Verlag, Berlin (1999)
12. Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating Answer Set Programming and Constraint Logic Programming. Annals of Mathematics and Artificial Intelligence (2008)
13. Smith, B.M.: Handbook of Constraint Programming, chap. 11. Modelling, pp. 377–406. Foundations of Artificial Intelligence, Elsevier (2006)