Computers and Chemical Engineering 34 (2010) 2129–2148

Contents lists available at ScienceDirect

Computers and Chemical Engineering

journal homepage: www.elsevier.com/locate/compchemeng

Reactive scheduling framework based on domain knowledge and constraint programming

Juan M. Novas, Gabriela P. Henning*

INTEC (Universidad Nacional del Litoral – CONICET), Güemes 3450, S3000GLN, Santa Fe, Argentina

ARTICLE INFO

Article history:
Received 16 February 2010
Received in revised form 5 July 2010
Accepted 8 July 2010
Available online 17 July 2010

Keywords:
Reactive scheduling
Batch plants
Decision support systems
Knowledge-based scheduling
Constraint programming

ABSTRACT

Industrial environments frequently face disruptive events. This contribution presents a support framework, aimed at addressing the repair-based reactive scheduling problem. It is based on an explicit object-oriented domain representation and a constraint programming (CP) approach. When an unforeseen event occurs, the framework captures the in-progress agenda status, as well as the event effect on it. Based on this information, a rescheduling problem specification is developed. Tasks to be rearranged are recognized and the set of the most suitable rescheduling action types (e.g. shift-jump, reassign, freeze) is identified. Since a given specification may lead to several solutions, the second stage relies on a CP model to address the problem just defined. To create such model, action types are automatically transformed into constraints. Provided that good quality schedules can be reached in low CPU times, alternative solution scenarios focusing on stability and regular performance measures can be posed for each problem.

1. Introduction

Predictive scheduling techniques deal with the generation of production plans while assuming stationary operation conditions along the whole scheduling horizon. However, real industrial environments are dynamic in nature and unforeseen events frequently disrupt the in-progress schedule. For instance, new order arrivals, orders' cancellations/modifications, unit breakdowns, changes in batch processing/setup times, late arrivals of raw materials, etc., are some of the different kinds of unexpected events that are faced in industry on a daily basis.

As Smith (1995) has pointed out, in most industrial plants, scheduling is an ongoing reactive process where evolving and changing circumstances continually force reconsideration and revision of pre-established plans. The performance of an industrial environment ultimately hinges on the management's ability to rapidly adapt schedules to fit changing circumstances over time. To facilitate this task, and make it fast and handy, rescheduling procedures are needed. However, current computer-based scheduling systems deal poorly (if at all) with this need (Kelleher & Cavichiollo, 2001), and much work still needs to be done to address this problem. Ouelhadj and Petrovic (2009) provide an updated review of the state-of-the-art of research on dynamic scheduling of manufacturing facilities. Besides, they introduce the principles of several dynamic scheduling techniques (i.e. dispatching rules, heuristics and meta-heuristics, artificial intelligence-based methods, multi-agent systems, etc.) and compare their relative merits.

Despite the great practical interest of the problem not many proposals exist in the process systems engineering open literature. The scheduling research community has largely neglected this problem, focusing instead on the generation of optimal static schedules. However, in the last two decades some reactive scheduling methodologies have been reported and they have been recently reviewed by Li and Ierapetritou (2008). The earliest proposals were based on algorithmic and heuristic procedures. Cott and Macchietto (1989), who considered fluctuations of processing times as the disruptive event, presented a shifting algorithm to modify the starting times of processing steps by the maximum deviation between the expected and actual processing times of all related processing steps. Shortly, Hasebe, Hashimoto, and Ishikawa (1991) proposed another algorithmic approach to deal with the reactive scheduling of multi-product batch plants. They allowed two reordering operations, the insertion of a job and the exchange of two jobs, and pointed out that simultaneous reordering of several jobs would be advantageous, but quite demanding from a computational point of view. Later, Kanakamedala, Reklaitis, and Venkatasubramanian (1994) addressed the problem of reactive scheduling of multipurpose batch plants in the event of unexpected deviations in processing times and unit availabilities. They proposed a least impact heuristic approach that allowed time shifting and unit reallocation. Huercio, Espuña, and Puigjaner (1995) and Sanmartí, Huercio, Espuña, and Puigjaner (1996) presented other heuristic-based methodologies to tackle problems of variations in task processing times and equipment availability. Two rescheduling strategies were proposed:

* Corresponding author. Tel.: +54 342 4559175x2102; fax: +54 342 4550944.
E-mail address: ghenning@intec.unl.edu.ar (G.P. Henning).

0098-1354/$ – see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.compchemeng.2010.07.011

Nomenclature

Objects

$Product_p$	instance of class *Product*
$Recipe_r$	instance of class *Recipe*
$ProcessingTask_t$	instance of class *ProcessingTask*
$Unit_u$	instance of class *Unit*
$Stage_s$	instance of class *ProcessStage*
$Batch_i$	instance of class *Batch*
$Task_{(i,s)}$	instance of class *Task* associated with the $Batch_i$ and $Stage_s$ instances
$Event_e$	instance of any subclass of *DisruptiveEvent*

Task sets

T_e	tasks pertaining to the on-going agenda when $Event_e$ takes place
T_e^{AE}	tasks which are already executed at the rescheduling time point associated with $Event_e$
T_e^{IP}	tasks which are being executed (in-progress tasks) at the rescheduling time point associated with $Event_e$
T_e^{NE}	tasks which are not yet executed at the rescheduling time point associated with $Event_e$
T_e^{CA}	tasks to be cancelled due to the occurrence of $Event_e$
T_e^{NW}	new tasks to be inserted in the active schedule due to the occurrence of $Event_e$
RT_e	tasks to be considered in the rescheduling problem triggered by $Event_e$
NT_e	tasks non-involved in the reactive process associated with $Event_e$
RT_e^{DA}	tasks to be considered in the rescheduling problem which are directly affected by $Event_e$
RT_e^{IA}	tasks to be considered in the rescheduling problem which are indirectly affected by $Event_e$
RT_e^{NA}	tasks to be considered in the rescheduling problem which are not affected by $Event_e$

CP model sets/indices

I/*i, i'*	batches
S/*s, s'*	stages
U/*u,u'*	processing units
$U_{(i,s)}$	subset of equipment items in which $Task_{(i,s)}$ can be executed
RT_e^A	set of tasks pertaining to the rescheduling problem that arises due to the occurrence of $Event_e$ which are to be assigned (new tasks) or reassigned.
RT_e^S	set of tasks pertaining to the rescheduling problem that arises due to the occurrence of $Event_e$ that maintain their current unit assignment and can be locally rearranged, being shifted or changing their position in the unit sequence.
RT_e^F	set of tasks pertaining to the rescheduling problem that arises due to the occurrence of $Event_e$ which are to be frozen; i.e. they maintain their current unit assignment and start time.
CU_u	equipment items which are connected with $Unit_u$

CP model parameters

dd_i	due-date of batch *i*
$est_{(i,s)}$	earliest start time of $Task_{(i,s)}$
$ls_{(i,s)}$	length of the left shift (reordering) interval of $Task_{(i,s)}$
$pt_{i,s,u}$	processing time of batch *i* on stage *s* when executed on $Unit_u$
rd_i	ready time of batch *i*
rd_u	ready time of $Unit_u$
$rs_{(i,s)}$	length of the right shift (reordering) interval of $Task_{(i,s)}$
$rt_{(i,s)}$	release time of $Task_{(i,s)}$

CP model variables

$Task_{(i,s)}.start$	start time of $Task_{(i,s)}$
$Task_{(i,s)}.end$	finish time of $Task_{(i,s)}$
$Task_{(i,s)}.duration$	duration of $Task_{(i,s)}$
Mk	Makespan
TD	total deviation of task start times

Performance measures

TCT	total completion time
TT	total tardiness
NES	normalized equipment stability
NST	normalized number of temporal shifted tasks

Shifting of task processing times and reassignment of tasks to alternative units. Their method generates a set of decision trees using alternative unit assignments, each based on a conflict in the real production schedule caused by a deviation from the nominal one.

More recent works addressed the reactive scheduling problem through mathematical programming approaches that mostly rely on Mixed Integer Linear Programming (MILP) formulations. Vin and Ierapetritou (2000) considered the rescheduling of multiproduct batch plants in the event of two types of disturbances: equipment breakdown and rush order arrival. They applied the continuous-time formulation originally proposed by Ierapetritou and Floudas (1998) and reduced the computational effort by fixing those binary variables involved in the period before the unforeseen event occurs. Roslöf, Harjunkoski, Björkqvist, Karlsson, and Westerlund (2001) proposed an MILP-based heuristic approach that can be employed to improve a non-optimal schedule or to update the in-progress schedule of a facility having a single/critical processing unit. In this work rescheduling is performed by iteratively releasing and reallocating a small number of jobs from the current schedule. Computational complexity is controlled by limiting the size of the set of jobs to be reallocated into the schedule. Méndez and Cerdá (2003) employed different rescheduling operations (i.e. start time shifting, local reordering, unit reallocation and insertion of new batches) to carry out the reactive scheduling of a multiproduct batch plant having a single/critical processing stage with several units operating in parallel. This work was extended in Méndez and Cerdá (2004) to consider multistage multiproduct facilities having limited availability of discrete renewable resources. Both approaches rely on a predictive scheduling formulation that was previously proposed by the same authors.

Janak, Floudas, Kallrath, and Vormbrock (2006) have also addressed this kind of problem. They presented an MILP-based method to respond to unexpected events, such as equipment breakdown and addition/modification of orders. To avoid full rescheduling, the approach identifies tasks which are not affected by the event and can be carried out as originally scheduled. The resulting tasks, along with additional subsets of tasks, are then fixed. The rest of horizon is rescheduled using an efficient MILP mathematical framework, developed for short-term scheduling problems, and adapted to reflect the effects of the unpredicted event. More recently, Kopanos, Capón-García, Espuña, and Puigjaner (2008) have stressed the need to take rescheduling costs into account in the formulations of reactive scheduling mathematical models. By introducing such costs in the objective function, schedules that are more stable and easier to implement

were generated. They were developed with the formulation of Méndez and Cerdá (2003). The trade-off between the original objective function optimization and the smooth operation of the plant was carefully analyzed.

This contribution tackles the reactive scheduling problem by proposing a support environment and a methodology that relies on both, an explicit representation of the domain knowledge and a Constraint Programming (CP) approach. The goal of the proposal is to provide immediate responses to disruptions, while introducing minimum changes to the on-going schedule in order to maintain a smooth operation of the plant. The approach acknowledges that most of the already taken scheduling decisions need to remain the same or at most they should experience a limited number of changes. The paper is organized as follows. In Section 2, the main concepts about reactive and dynamic scheduling problems are reviewed. Section 3 provides an overview of the proposed framework, Section 4 presents the domain knowledge representation as well as the problem characterization and specification on which the proposal relies, and Section 5 describes the constraint programming model. Finally, in Section 6 we illustrate the capabilities of the approach by means of several example problems, and close in Section 7 with conclusions and some directions for future work.

2. An overview of dynamic scheduling problems and their elements

As mentioned before, the problem of scheduling in the presence of real-time disruptive events has received limited attention. Not many surveys have been done in this area and very few works have tried to understand and organize research contributions. Vieira, Herrmann, and Lin (2003) have made a remarkable effort to systematize knowledge in this domain. Their work presents definitions and concepts appropriate for most applications of rescheduling manufacturing systems and also describes a framework for understanding rescheduling strategies, policies and methods.

According to Ouelhadj and Petrovic (2009), the problem of scheduling in the presence of real-time events is termed *dynamic scheduling*. These authors have classified dynamic scheduling problems under four categories: Completely reactive scheduling, predictive-reactive scheduling, robust predictive-reactive scheduling and robust pro-active scheduling. This classification is related to what Vieira et al. (2003) have referred as the adopted strategy: whether or not production schedules are generated. In *completely reactive scheduling* no firm schedule is generated in advance, decisions are made locally in real-time, and priority dispatching rules are frequently used. On the other hand, *predictive-reactive scheduling*, the most common approach used in manufacturing systems, is a process under which schedules are revised in response to real-time events. Many predictive-reactive scheduling strategies are based on simple adjustments that consider only shop efficiency; however, the new schedule may deviate significantly from the original one. To overcome this problem, *robust predictive-reactive scheduling* focuses on building schedules that simultaneously take into account both shop efficiency and deviation from the original schedule (stability). Finally, *robust pro-active scheduling* approaches focus on building predictive schedules that satisfy performance requirements predictably in a dynamic environment (Vieira et al., 2003). This category has also been referred as *robust scheduling* by some authors like Aytug, Lawley, McKay, Mohan, and Uzsoy (2005). They have pointed out that this approach can be viewed as a form of under-capacity scheduling, where the amount of work scheduled in a time period is based on the historical performance of the equipment.

2.1. Rescheduling policies and methods

The second and the third category presented above correspond to what is generally known as *rescheduling*; the process of updating an existing production schedule in response to disruptions or other changes. Rescheduling needs to address various issues: (i) when and how to react to real-time events, and (ii) the method or methods' combination used to revise the existing schedule. According to Vieira et al. (2003) a *rescheduling policy* specifies when and how rescheduling is done.

Regarding the first issue, when to reschedule, three policies have been proposed in the literature: periodic, event-driven, and hybrid. In the *periodic policy*, schedules are generated at regular intervals, gathering all the available information from the shop floor at the planned rescheduling time points. The dynamic scheduling problem is decomposed into a series of static problems that can be solved by resorting to classical predictive scheduling approaches. Under an *event-driven policy,* rescheduling is triggered in response to an unforeseen event. This policy runs the risk of initiating rescheduling activities in the face of events not causing significant disruptions; thus, it may expend computational resources needlessly and potentially cause unnecessary changes in the current production schedule. In contrast, the drawback of the periodic policy is that it ignores events taking place between rescheduling points, which in some cases may lead to unfeasible agendas. Hence a combination of these two approaches under a *hybrid policy* appears to be attractive. It resorts to a periodic rescheduling, but a rescheduling activity can also be invoked if a significant disruption is observed.

Regarding the second issue, how to react to unexpected events, the literature provides two main approaches: *schedule repair* and *complete rescheduling*. *Schedule repair* refers to some local adjustments of the current schedule, while *complete rescheduling* generates a new production schedule from scratch. Complete rescheduling might be better in reaching optimal solutions with respect to traditional scheduling objectives, but these solutions are rarely put in practice since they result in instability and lack of continuity in the detailed plant schedules. As a result, complete rescheduling leads to additional production costs and to what has been termed as *shop floor nervousness*. Furthermore, complete rescheduling might require prohibitive computational times when fast responses are mandatory. Due to these reasons schedule repair methods are preferred in practice; they preserve the stability of the system and demand few computational resources.

2.2. Event types

Industrial environments are dynamic in nature and are subject to continuous disturbances, referred as real-time events, which can change the status of the environment and affect its performance. If an event causes significant deterioration in performance, and an event-driven rescheduling policy is adopted, the event will trigger a rescheduling activity to reduce its impact. Such rescheduling activity takes place at a *rescheduling point*, which is the point in time when a schedule is created or revised. Generally, corrections are performed at or soon after the occurrence of the event. Thus, under an event-driven rescheduling policy each disruptive event is associated with a rescheduling point, which should be as close as possible to the time point where the event took place.

Real-time events have been classified into two main categories (Vieira et al., 2003): Resource-related and job-related. The first category includes machine breakdown, over- or underestimation of processing times, limited manpower, delay in the arrival or shortage of raw materials, defective material, etc. The second category includes rush jobs, due-date modification, change in job priority, job cancellation, batch reprocessing, etc. Independently of the

event category, there appear to be three key dimensions of unforeseen events, especially those related to resources: *cause*, *context*, and *impact* (Aytug et al., 2005). *Cause* can be viewed as a change of state of the affected resources, *context* is associated with the situation of the manufacturing environment at the moment the event takes place, and *impact* refers to its result on the manufacturing system. Context and impact are related among themselves; not all the jobs or situations react the same way to the same perturbation. For instance, a unit failure lasting one day does not have the same impact on a one-week schedule if it occurs at the beginning or at the end of the scheduling horizon. Similarly, a unit breakdown extending for twelve hours does not have the same consequences on a two-day schedule or on a schedule that lasts a week.

2.3. Performance measures

Predictive scheduling activities are generally guided by measures of schedule efficiency and/or cost. The first group usually corresponds to time-based measures like, makespan, mean tardiness, mean flow time, maximum lateness, average resource utilization, etc. Since time-based performance measures do not completely reflect the economic performance of the scheduling system, cost related objective functions have also been adopted. Issues like job-profitability, operating cost or work-in-process minimization, or costs of missing due-dates or starting jobs too early, are aspects that have also been selected by managers.

In order to assess the departure of the revised from the initial schedule, and in addition to regular performance measures, different appraisals of schedule change should be considered. A schedule not only provides an operational plan that allocates resources and defines task timings, but also serves as a base agenda for other activities (e.g. releasing raw material movement orders, planning shipping operations, etc.). Hence, deviations from the original schedule disrupt the secondary plans derived from it, creating floor shop nervousness. Thus, stability measures that are associated with a smooth operation of the plant become relevant. For instance, metrics like the starting time deviations between the tasks of the new and the original schedules or a measure of the sequence difference between these two schedules, have been introduced in the objective function to minimize the degree of disruption (Vieira et al., 2003). Defining a proper disruption metric is not straightforward; it requires considerable work and analysis if it is to be meaningful and useful. Nevertheless, on top of this difficulty, another challenge faced in rescheduling environments is to take into account both disruption metrics and regular performance measures, balancing efficiency and stability.

3. Reactive support environment

A rescheduling process is similar to the process of generating an initial schedule, but has more constraints and objectives attached to it. When addressing a rescheduling situation most of the objectives and basic constraints that define the original problem still apply; however, the partially executed schedule, as well as the perturbation or triggering event has also to be taken into account. This contribution presents advances in the development of a support framework able to represent this context knowledge and to use it in the generation of solutions to rescheduling problems. The cur-

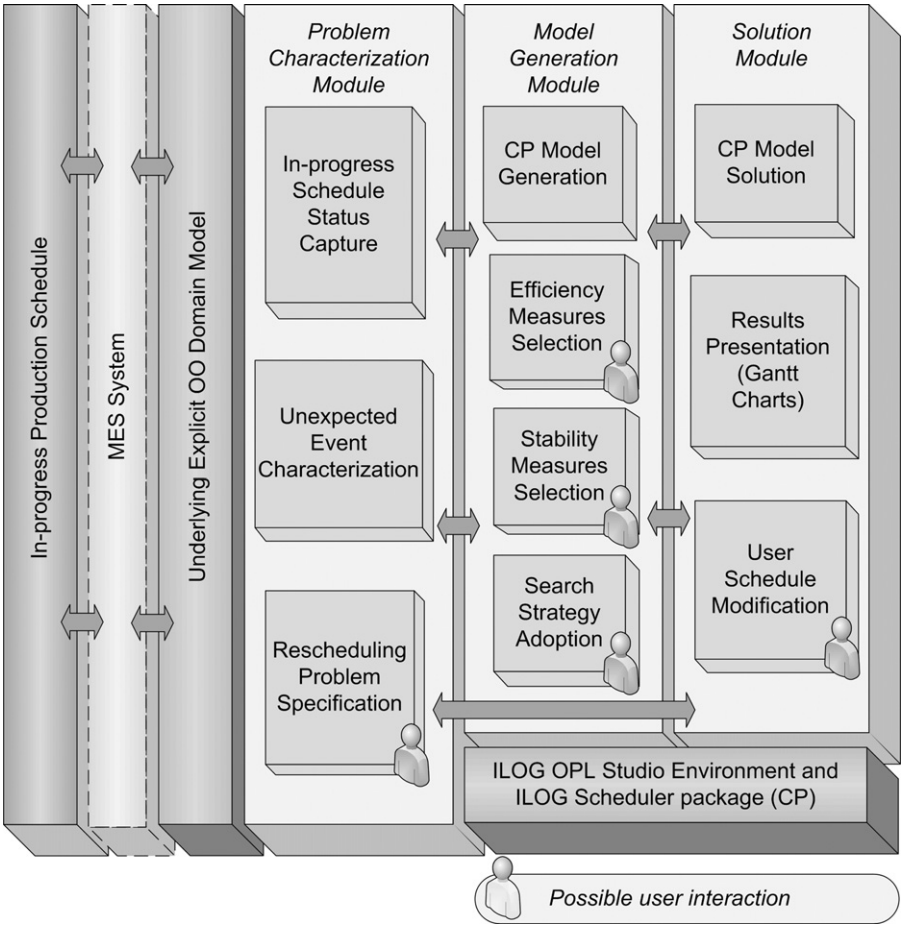

Fig. 1. Main components of the Reactive Scheduling Support Framework.

rent proposal is oriented towards multiproduct multistage batch plants, which operate under a batch-based approach, and with UIS, NIS/UW, NIS/ZW interstage storage and operational policies. It is limited to a set of unforeseen events (unit failure and performance modification, batch cancellation/modification/new arrival, batch quality problems). However, the underlying rationale of the system would allow extending it in the future to consider multipurpose batch plants, other interstage and operational policies and a wider range of disruptive events (e.g. modification of non-renewable resource availability).

The framework has been envisioned to operate under an event-driven rescheduling policy. The proposed approach assumes that the event which triggers the rescheduling process causes significant disruptions and rescheduling is actually necessary. The framework explicitly captures the status of the in-progress schedule, and typifies the unexpected event in order to characterize its context and impact. This allows making a proper specification of the rescheduling problem to be faced. This specification is then translated into a CP model and, finally, the resulting formulation is solved. The proposed solution methodology attempts to get together the benefits of a repair-based method (limited schedule disruption and low computational requirements), with the advantages of a complete rescheduling approach (non-myopic, overall view of the rescheduling system). Furthermore, one of its goals is to develop several alternative solutions in very low CPU times and to select the preferred one with reference to a set of objectives that measure both schedule efficiency and stability. The environment integrates different modules, as it is shown in Fig. 1, all based on an explicit domain representation. The various responsibilities assumed by these modules in relation to the solution methodology, which is presented in detail in the following sections of the paper, is depicted in Fig. 2.

The first phase of the approach (*Problem Characterization Module* in Fig. 1), uses information about the current status of the in-progress schedule (*Capture in-progress schedule status* step in Fig. 2), the event type, and its intrinsic characteristics (*Characterize event impact* step in Fig. 2), to automatically identify and classify tasks to be taken into account in the rescheduling problem, and to distinguish them from those not involved in the reactive process (*Identify tasks to be rescheduled* step in Fig. 2).

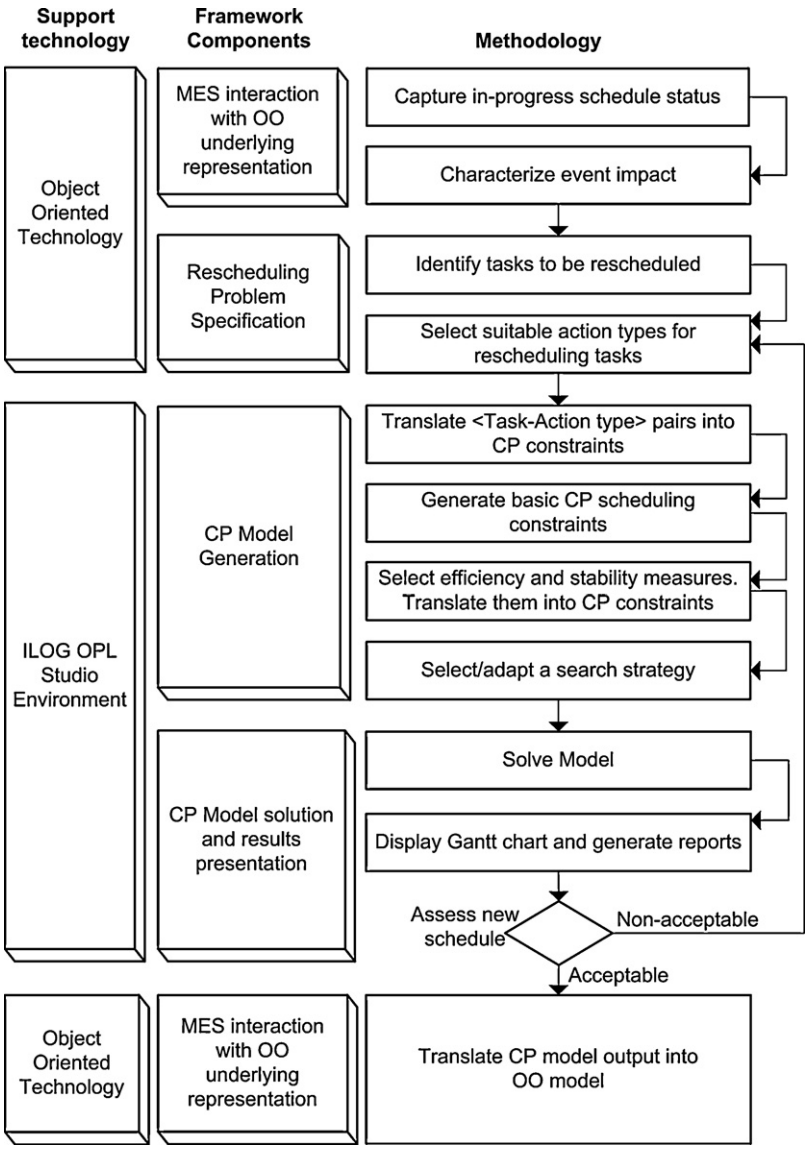

Fig. 2. Main steps of the solution methodology.

Tasks to be considered in the reactive problem are further categorized into directly-affected, indirectly affected and non-affected tasks; and for each of them, the most suitable rescheduling action type or action is specified (*Select suitable action types for rescheduling tasks* step in Fig. 2). An action type not only prescribes a type of repair operation to be applied (e.g. shift, reassign, etc.) on a processing task that can be subject to rescheduling, but also a range of possibilities for its application, e.g. feasible equipment for a reassignment operation, shifting interval, etc. Thus, an action type should not be interpreted as an instance of a local repair action, but as a generic specification of a set of operations.

A rescheduling problem is considered to be specified when for each relevant task an action type or an action is identified. For a given specification, due to the characteristics of the action types, many solutions could exist. Then, the second phase of the methodology relies on a CP approach that is in charge of instantiating the set of action types and generating repaired schedules (see *Model Generation Module*). In order to develop a solution this module needs to (i) transform the problem specification into a CP model, which entails both translating the identified action types into CP constraints and generating the basic model constraints (*Translate < Task-action type > pairs into CP constraints* and *Generate basic CP scheduling constraints* steps in Fig. 2), (ii) select an appropriate set of efficiency and stability performance measures (*Select efficiency and stability measures* step in Fig. 2), and (iii) to adopt a search strategy based on domain knowledge, in case of needing it (*Select/adapt a search strategy* in Fig. 2). Finally, the solution of the CP model and its associated search strategy will render the repaired schedule, in which the proposed repair action types will be instantiated.

4. Domain knowledge representation and problem characterization

Scheduling is a knowledge intensive activity in terms of domain information. The explicit representation of this knowledge type is a critical issue in any scheduling support system, especially if it is to be integrated with other enterprise support systems (Henning, 2009). Besides, a major function of a production schedule, which is often overlooked by the research community (Aytug et al., 2005), is that of providing information visibility for the rest of the organization, and for internal and external suppliers and customers. The more explicit the information of a schedule is, the more visible it becomes. Furthermore, in the context of this proposal, an explicit representation of a manufacturing environment and its production plan (see *Underlying Explicit OO Domain Model* in Fig. 1) would allow making informed decisions along the various steps of the solution methodology.

Fig. 3 presents an UML class diagram, which is one of the diagrams of the object-oriented technology (Booch, Rumbaugh, & Jacobson, 1999) that provides a simplified version of the model of the multiproduct batch plant scheduling knowledge included in the framework. This information is organized into different conceptual perspectives; resource, recipe and schedule views, which are described below. To support compatibility with other applications–e.g. Manufacturing Execution Systems (MESs), Advanced Planning Systems (APS), etc.–, the proposed representation was developed based on the ISA-S88 standard (ANSI/ISA, 1995; ANSI/ISA, 2001a, ANSI/ISA, 2001b; ANSI/ISA, 2003).

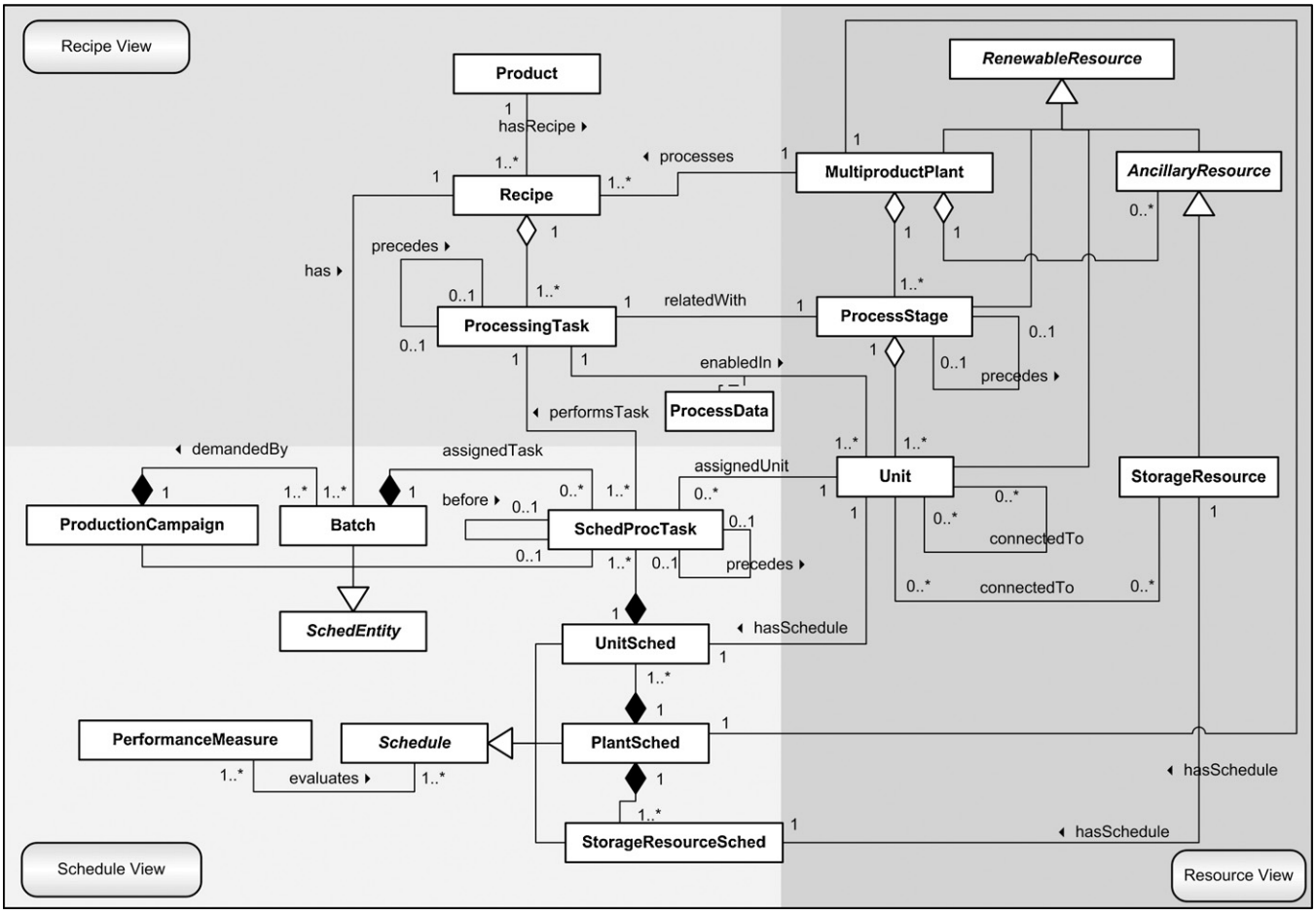

Fig. 3. Partial view of batch scheduling domain concepts.

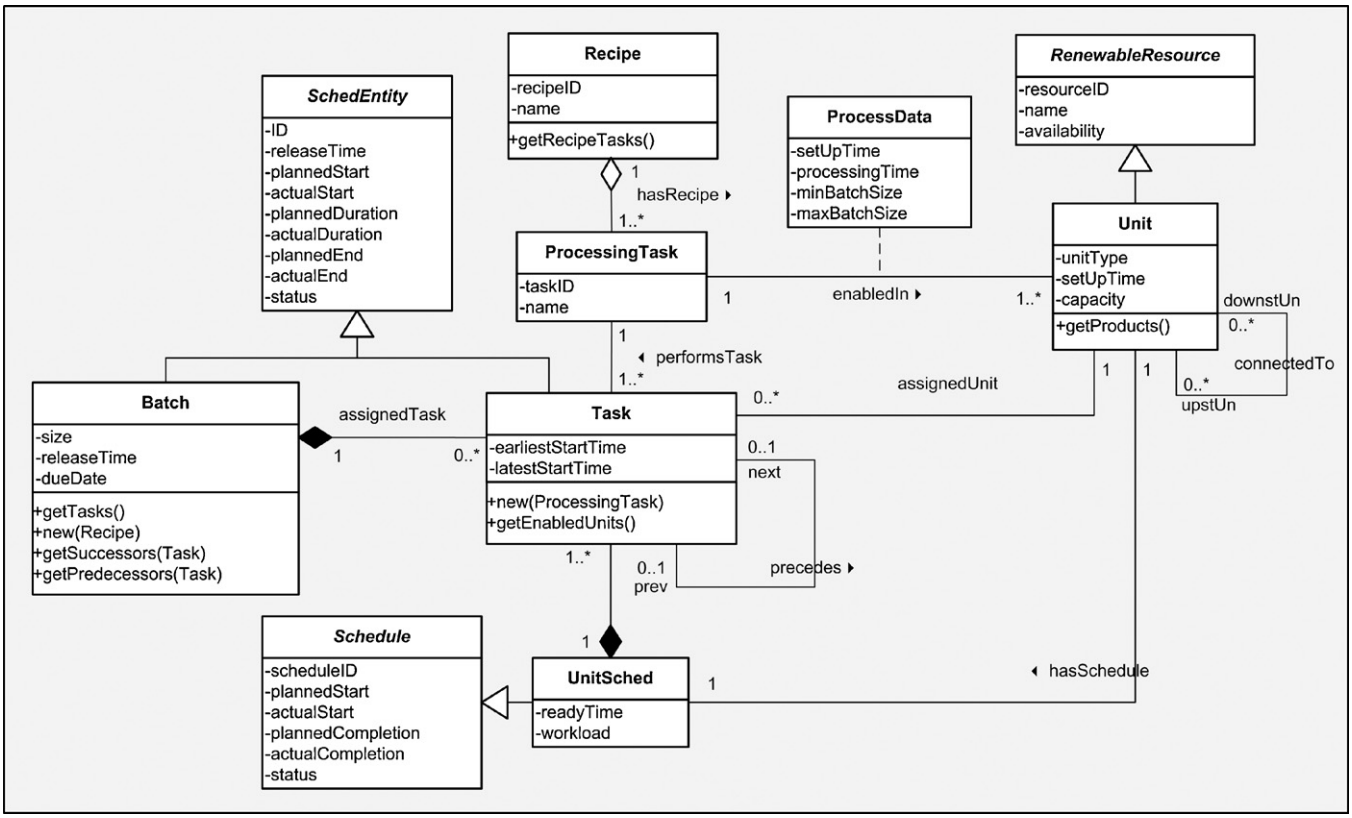

Fig. 4. Partial view of some relevant entities and their properties.

The *resource view* represents the physical resources of the plant and their relationships, but also has links with elements pertaining to the other views. The *Resource* class is specialized into the renewable and non-renewable resource (See *RenewableResource* class in Fig. 3). A multiproduct plant is modeled as a kind of renewable resource, which is an aggregation of process stages (*ProcessStage* class) and ancillary equipment. In turn, a process stage is modeled as an aggregation of processing units (*Unit* class in the diagram). This view also includes an explicit representation of the pipeline connections between units and between units and storage resources (see *connectedTo* associations in Fig. 3).

The *recipe view* models, for each product, the set of processing activities (*ProcessingTask* class) required to produce a batch of it, as well as the precedence relationships among them. As shown in Fig. 3, this perspective is also related to the other views. For instance, the *ProcessData* association class, linking *Unit* and *ProcessingTask*, takes care of representing manufacturing data.

Finally, the *schedule view* models those entities involved in scheduling and rescheduling activities and their associations. A production campaign (*ProductionCampaign* class) is composed of a set of batches (*Batch* class). The execution of a batch leads to a set of tasks (see *assignedTask* composition link between the classes *Batch* and *SchedProcTask*). The set of *SchedProcTask* instances that are assigned to a particular unit comprise the schedule of such unit (*UnitSched* class). Likewise, the set of unit and storage resource schedules embrace the plant schedule (see composition links of the *PlantSched* class). Moreover, since operational plans are always assessed, the *Schedule* entity has several performance measures associated with it. The *PerformanceMeasure* class is specialized to represent the various efficiency, cost, and stability metrics that allow assessing a schedule (specialization not shown for simplicity reasons). These classes contain methods that define the corresponding metrics.

Fig. 3 shows that the *SchedProcTask* instances are linked among themselves by the *precedes* and *before* associations. The first relation links task entities comprising the same batch and it has the same meaning than the *precedes* association between processing tasks that belong to a given recipe. On the other hand, the *before* link establishes a direct precedence relation between two tasks that comprise the schedule of a particular unit. These additional constraints about the objects on the model, as well as others that are required to avoid ambiguities or inconsistencies have been expressed in the Object Constraint Language (OCL), OMG (2006). They have not been included due to lack of space.

The class diagram of Fig. 4 presents a simplified view of the domain knowledge, showing some of the properties and methods of the most relevant entities. Most of the attributes and methods that are going to be employed in the specifications presented in Sections 4.1 and 4.2 are included in this diagram. Please note that in this figure, the name of the *SchedProcTask* entity has been changed to *Task*, and from now on, for simplicity reasons, it is going to be named in this way. Not only static information of resources (*Resource view* in Fig. 3) need to be explicitly represented, but also their temporal attributes are important too. As shown in Fig. 4, all scheduled entities have attributes (e.g. *actualStart*, *actualEnd*, *actualDuration*, etc.) whose values can be updated by the MES system at any time point during the scheduling horizon, and from which the status of the scheduling objects can be inferred.

Events and their specific characteristics are other critical aspects of the problem; therefore, the different types of events and their associated knowledge need to be explicitly captured by the underlying explicit OO representation. The event classification adopted in this proposal follows the one that was already described in Section 2.2. This model view, shown in Fig. 5, is also associated with the other domain elements. The attributes that appear in this class dia-

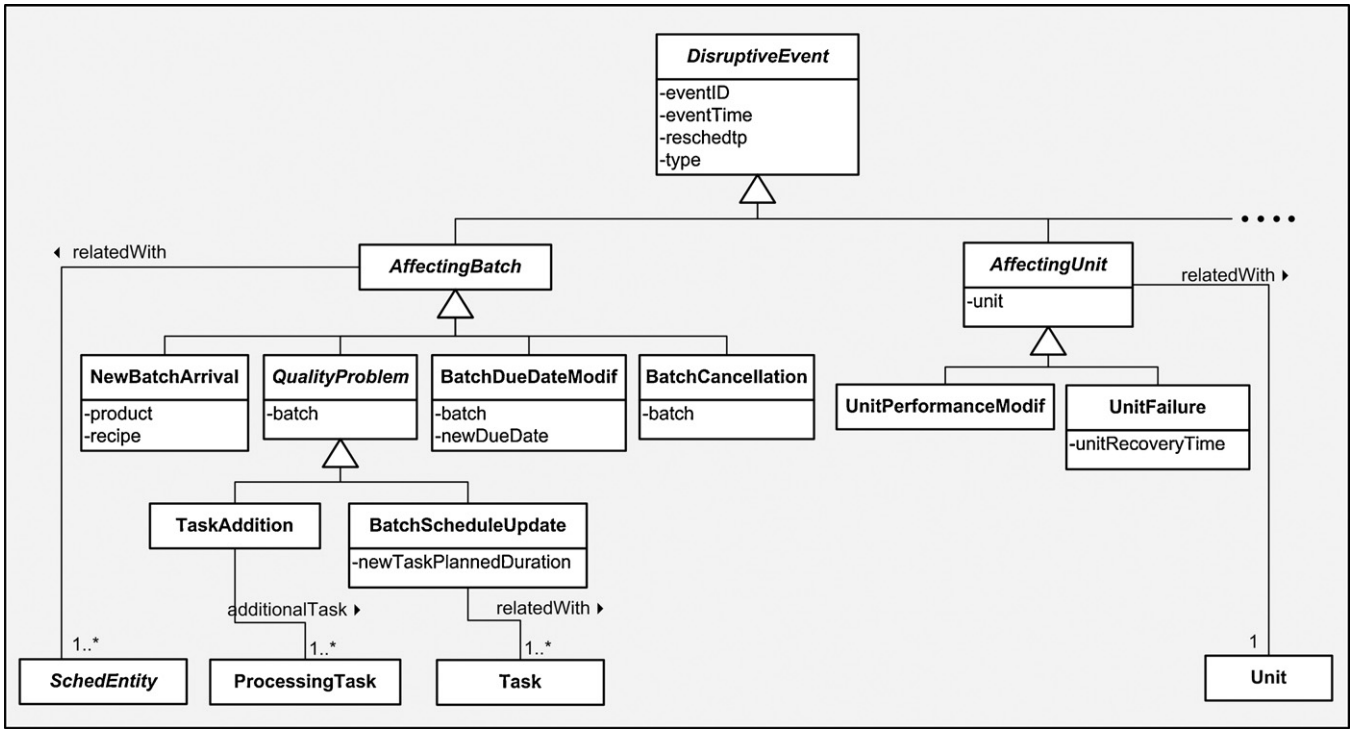

Fig. 5. Disruptive events representation.

gram are the ones that are employed in the specifications included in Sections 4.1 and 4.2.

Having the domain knowledge explicitly captured, when an unforeseen event occurs, it can be characterized (*Unexpected Event Characterization* module in Fig. 1) and the status of the current schedule at the rescheduling point can be determined (see *In-progress Schedule Status Capture* module in Fig. 1). Thus, both the context and the impact of the event can be properly identified, and, from them, the rescheduling problem can be specified (*Rescheduling Problem Specification* module). More details about these modules and their associated solution steps are presented in the following sections.

4.1. Capture of the current operational plan and its status at the rescheduling point

When addressing a rescheduling situation triggered by a particular event (an instance of the *DisruptiveEvent* class of Fig. 5), referred from now onwards as $Event_e$, the status of each task involved in the in-progress schedule is supposed to be known and the tasks' attributes are assumed to have updated values, as a result of continuous communication with a MES. At the rescheduling point associated with the event, denoted as $Event_e.reschedtp$, the *Problem Characterization Module* carries out a decision-making activity about the tasks to be included in the reactive process, which only includes the non-executed tasks, as well as those that need to be carried out again. On the contrary, the already executed ones are left completely aside; thus, there is no need to artificially fix their associated variables, as in Vin and Ierapetritou (2000) or Janak et al. (2006). Nevertheless, since these entities remain represented in the system, their associated information can be accessed, whenever necessary, to define the release times of units and/or tasks which are still to be processed.

The set of tasks pertaining to the on-going agenda, named T_e, are first classified into three different subsets, independently of the type of unforeseen event, but depending on their status at the rescheduling time point (*RTP*). These groups are:

(i) Set of already executed tasks at the *RTP* associated with $Event_e$, which are denoted as T_e^{AE}, and defined by expression (1).
(ii) Set of non-executed tasks at the *RTP* associated with $Event_e$, which are identified as T_e^{NE}, and defined by expression (2).
(iii) Set of in-progress tasks at the *RTP* associated with $Event_e$, which are referred as T_e^{IP}, whose elements are captured by expression (3).

In all these expressions, and in the remaining of this paper, an instance of the class *Task* that stands for the processing of batch i at stage s is represented in a simplified way as $Task_{(i,s)}$.

$$\begin{aligned}&\text{ForAll}\,(Task_{(i,s)}|Task_{(i,s)}.actualEnd \le Event_e.reschedtp \Rightarrow\\ &\quad Task_{(i,s)} \in T_e^{AE} \wedge Task_{(i,s)}.status = \text{“}AlreadyExecuted\text{”})\end{aligned} \tag{1}$$

$$\begin{aligned}&\text{ForAll}\,(Task_{(i,s)}|Task_{(i,s)}.plannedStart \ge Event_e.reschedtp \Rightarrow\\ &\quad Task_{(i,s)} \in T_e^{NE} \wedge Task_{(i,s)}.status = \text{“}NonExecuted\text{”})\end{aligned} \tag{2}$$

$$\begin{aligned}&\text{ForAll}\,(Task_{(i,s)}|(Task_{(i,s)}.actualStart \le Event_e.reschedtp \wedge\\ &\quad Task_{(i,s)}.plannedEnd \ge Event_e.reschedtp \Rightarrow\\ &\quad Task_{(i,s)} \in T_e^{IP} \wedge Task_{(i,s)}.status = \text{“}InProgress\text{”})\end{aligned} \tag{3}$$

In addition, if the type of disruptive event is the cancellation of a particular batch instance, identified as $Batch_i$, all its associated tasks which have not been yet executed at the *RTP*, are going to be included in the set of cancelled tasks, which is identified as T_e^{CA}, and has members defined by expression (4).

$$\begin{aligned}&\text{Exists}\,(Event_e.type = \text{“}BatchCancellation\text{”} \wedge\\ &\quad Event_e.batch = Batch_i),\\ &\text{Let}\,BatchTaskSet \leftarrow Batch_i.getTasks(\,)\\ &\quad \text{ForAll}\,(Task_{(i,s)}|Task_{(i,s)} \in BatchTaskSet \wedge\\ &\quad Task_{(i,s)}.plannedStart \ge Event_e.reschedtp \Rightarrow\\ &\quad\quad Task_{(i,s)} \in T_e^{CA} \wedge Task_{(i,s)}.status = \text{“}Cancelled\text{”})\end{aligned} \tag{4}$$

Moreover, if one of the processing tasks of the cancelled $Batch_i$ is actually being executed at the *RTP*, such task is immediately

shut down and considered as already executed. This condition is captured by expression (5).

$$\begin{aligned}&(\text{Exists}\,(Event_e.type = \text{“}BatchCancellation\text{”}\wedge\\ &\quad Event_e.batch = Batch_i)\wedge\\ &\text{Exists}\,(Task_{(i,s)}|Task_{(i,s)}.actualStart \le Event_e.reschedtp\wedge\\ &\quad Task_{(i,s)}.plannedEnd \ge Event_e.reschedtp)) \Rightarrow\\ &\qquad Task_{(i,s)} \in \mathrm{T}_e^{AE} \wedge Task_{(i,s)}.actualEnd = Event_e.reschedtp\wedge\\ &\quad Task.status = \text{“}AlreadyExecuted\text{”}\end{aligned} \tag{5}$$

Finally, new tasks to be added to the schedule due to the occurrence of an $Event_e$, corresponding to the rush arrival of a new batch, belong to the set T_e^{NW}. For each new batch $Batch_i$, the number of tasks to be included in the set T_e^{NW} is equal to the number of processing tasks defined in the recipe of the batch product (see expression (6) and Fig. 4). These tasks are going to be included in the system as instances of the *Task* class. The same occurs with a batch that is actually being processed in a unit that suffers a breakdown and the batch cannot continue its normal execution, requiring to be redone. In this case, the batch's already executed tasks, as well as its in-progress task, would need to be carried out again.

$$\begin{aligned}&\text{Exists}\ (Event_e.type = \text{"}NewBatchArrival\text{"} \wedge Event_e.product = Product_p \wedge\\ &\qquad Event_e.recipe = Recipe_r),\\ &\quad \text{Let}\ \ Batch_i \leftarrow Batch.new(Recipe_r)\\ &\qquad ProcessingTaskSet \leftarrow Recipe_r.getRecipeTasks(),\\ &\qquad \text{ForAll}\,(ProcessingTask_t\,|ProcessingTask_t \in ProcessingTaskSet,\\ &\qquad\quad \text{Let}\ \ Task_{(i,s)} \leftarrow SchedProcTask.new(ProcessingTask_t),\\ &\qquad\quad Task_{(i,s)} \in \mathrm{T}_e^{NW})\end{aligned} \tag{6}$$

The sets defined in the preceding paragraphs are related in the following way:

$$\mathrm{T}_e = \{\mathrm{T}_e^{AE} \cup \mathrm{T}_e^{IP} \cup \mathrm{T}_e^{NE}\};\quad \mathrm{T}_e^{CA} \subseteq \mathrm{T}_e^{NE};\quad \mathrm{T}_e^{NW} \notin \mathrm{T}_e \tag{7}$$

The previous classification allows distinguishing those tasks to be involved in the rescheduling process due to the occurrence of the unexpected event $Event_e$. These activities comprise the RT_e set and are defined by expression (8). The following sections will focus on tasks belonging to RT_e.

$$\begin{aligned}&\text{ForAll}\,(Task_{(i,s)}|Task_{(i,s)} \notin \mathrm{T}_e^{CA} \wedge (Task_{(i,s)} \in \mathrm{T}_e^{NE} \vee Task_{(i,s)} \in \mathrm{T}_e^{NW})\\ &\quad \Rightarrow Task_{(i,s)} \in \mathrm{RT}_e)\end{aligned} \tag{8}$$

The remaining tasks are those not involved in the reactive process; they comprise the NT_e set and are specified by means of expression (9).

$$\begin{aligned}&ForAll\,(Task_{(i,s)}|Task_{(i,s)} \in \mathrm{T}_e^{AE} \vee Task_{(i,s)} \in \mathrm{T}_e^{IP} \vee Task_{(i,s)} \in \mathrm{T}_e^{CA}\\ &\quad \Rightarrow Task_{(i,s)} \in \mathrm{NT}_e)\end{aligned} \tag{9}$$

In addition to the previous classification, the *Problem Characterization Module* uses information from the task instances and the disrupting event in order to specify other data that are also necessary to properly define the reactive scheduling problem launched at the *RTP*. Specifically, the module specifies: (i) the new ready times of the equipment items, (ii) the earliest and latest start times of those tasks that are associated with in-progress and new batches, etc. This information is required to properly specify the reactive scheduling problem, as discussed in the following section.

4.2. Rescheduling action types and actions

For each task belonging to the set RT_e, its associated rescheduling action type or action has to be chosen in order to generate the specification of the reactive scheduling problem to be faced and solved. As mentioned before, when a task is linked to an action type it does not have a specific and pre-defined local rescheduling operation related to it, but a generic action that prescribes a rescheduling operation category and a range of possibilities to apply it. The framework includes the following action types:

Shift-JumpAT(⟨*Task*⟩, *ls*, *rs*). This action type represents a generic reordering operation on the same equipment item, so the task remains allocated to the same unit where it was originally assigned, but the timing of its start time can change. Thus, it allows the task to modify its start time by pushing it forward (right-shift) or backwards (left-shift) within prescribed limits. However, it also allows moving the task to another position in the sequence associated with the unit schedule, thus performing a reordering in the sequence by means of a jumping move on the same unit. Any of these "shift-jump" movements can be done by allowing the task to place its start time at any point within the limits established by the following time window: [*Task.plannedStart* − *ls*; *Task.plannedStart* + *rs*], where *ls* stands for the allowed left shift (reordering) interval and *rs* for the right shift (reordering) one.

ReassignAT(⟨*Task*⟩, *SetOfUnits*). This action type stands for a generic reallocation operation of a particular task that can be reassigned to any of the equipment items that belong to the *SetOfUnits*. When this set includes the current equipment item, the task can remain assigned to it if such unit is available. By definition, *SetOfUnits* comprises all the equipment items where the processing task can be carried out (the ones retrieved by the *getEnabledUnits* method of the class *Task*—see model described in Section 4), which, in addition, should be operable within the scheduling period.

AssignAT(⟨*Task*⟩, *SetOfUnits*). This action type represents a generic assignment operation of a particular task that can be allocated to any of the equipment items that belongs to the *SetOfUnits*. By definition, *SetOfUnits* comprises all the equipment items where the processing task can be carried out and which are also operable within the scheduling period. *AssignAT* will be employed to allocate tasks that need to be redone/reprocessed or are associated with the insertion of new batches.

In principle, *ReassignAT* is more disruptive than a *Shift-JumpAT* action type, which is likely to render more stable schedules, since the assigned unit is kept and the start time modification is bounded. Both *ReassignAT* and *AssignAT* refer to allocations of tasks to units, which can occur at any time point during the reactive scheduling horizon. Though they are similar in essence, they are distinguished because their task argument is different. *ReassignAT* has as argument a scheduled, but non-executed task, while *AssignAT* is applied to a new task. In addition to the previous action types the framework includes the following phantom action, which is not a generic one:

Freeze(⟨*Task*⟩). As its name indicates this operation applies to a particular task not allowing it to be moved to another unit, neither to modify its start time and duration; thus, the task is frozen in its current position. This action concerns those tasks which for different reasons must retain their current unit assignment as well as start and end times (e.g. tasks that are about to start in a non-disrupted unit when a unit failure occurs in another one, and already have their associated raw materials in place to start the manufacturing process).

It should be remarked that any task linked to an action type is bound to a set of rescheduling alternatives independently of the sta-

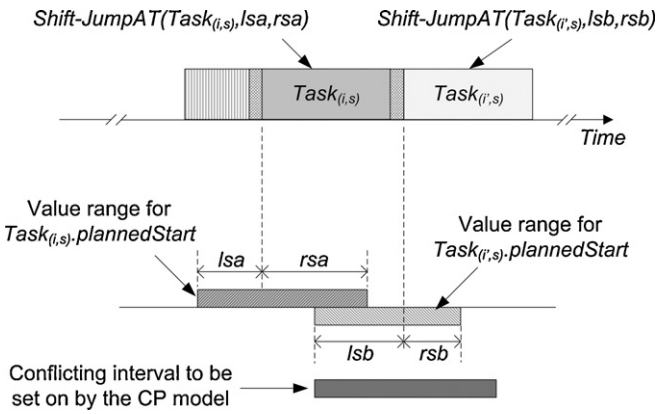

Fig. 6. Start time temporal intervals corresponding to the *Shift-JumpAT* action type associated with two tasks that belong to the RT_e set, showing the possible conflict of the tasks' execution times.

tus of the other scheduled activities and the action types assigned to them. For instance, a *Shift-JumpAT* that applies to a particular task allocated to a given unit, which prescribes a possible range of values for its start time, does not take into account the fact that there can be other tasks, assigned to the same unit, which can have associated *Shift-JumpATs* that implicitly define start time temporal intervals that might be in conflict. These potential conflicts among the temporal intervals are not considered at this stage and will be set on when solving the CP model. Fig. 6 depicts a simple example of a conflicting situation that could arise when defining shift-jump action types for two neighbour tasks. Thus, the problem specification is aimed at providing non-tight, but flexible repair actions, having a broader scope than the local repair operations presented in the literature. The goal is to give each task associated with a rescheduling action type several rescheduling alternatives, without considering at this stage, potential conflicts among the various task executions.

4.3. Rescheduling problem specification

In order to define the specific rescheduling action type to be associated with each member of set RT_e when a disruptive event $Event_e$ takes place, tasks are classified as:

(i) Tasks directly affected by the event, comprising the subset identified as RT_e^{DA}.
(ii) Tasks indirectly affected by the event, represented by the RT_e^{IA} subset, and
(iii) Tasks that are non-affected, which are included in the RT_e^{NA} subset.

This classification attempts to take into account two important elements, the event impact and the manufacturing environment context. Thus, the members of the different task subsets depend on the event type, its intrinsic characteristics and on how tasks are affected by it. The set memberships might also depend on the scheduling context at the *RTP*. The following sections present the way in which the set members can be systematically identified for some types of events.

Directly affected tasks are associated with particular action types that are mandatory and depend on the type of event (e.g. *ReassignAT* action types are specified for those tasks which are directly affected by a unit failure, and in case of a batch rush arrival, *AssignAT* action types are prescribed for the tasks associated with the new batch to be inserted). On the other hand, the action types or actions associated with indirectly and non-affected tasks cannot be prescribed beforehand and are specified based on domain knowledge and the state of the manufacturing environment at the *RTP*. The underlying idea is to associate these tasks with the less disruptive action types/actions so as to avoid causing unnecessary changes and to preserve the schedule stability as much as possible. Then, for a given event, alternative scenarios can be defined, where tasks belonging to the indirectly and non-affected task sets can be related to different rescheduling action types.

4.3.1. Unit breakdown (UBD) event

It takes place when a unit fails unexpectedly, becoming unavailable for the processing of tasks during a certain period that can extend up to the unit recovery time or over the whole scheduling horizon.

4.3.1.1. UBD event – directly affected tasks. When a unit breaks down and the plant operates under an unlimited intermediate storage policy (UIS) or a non-intermediate storage, unlimited wait (NIS-UW) policy, the tasks that are directly affected by the event are the following:

(i) The ones belonging to the T_e^{NE} set which are assigned to the broken-down unit and start later than the *RTP* but earlier than the unit recovery time.
(ii) Those tasks associated with the batch that was currently being processed in the unit at the time it failed, if such batch could not be recovered. In this case, all the batch associated tasks are considered to be directly affected, even the already executed ones.

Expressions (10) and (11) allow the automatic identification of the members of the RT_e^{DA} set, according to the previous specification.

$$\begin{aligned}&\text{Exists } (Event_e.type = \text{"UnitFailure"}),\\ &\quad \text{ForAll}(Task_{(i,s)} \mid (Task_{(i,s)} \in T_e^{NE} \wedge Task_{(i,s)}.assignedUnit = Event_e.unit \wedge\\ &\qquad Event_e.reschedtp \le Task_{(i,s)}.plannedStart \wedge\\ &\qquad Task_{(i,s)}.plannedStart \le Event_e.unitRecoveryTime)\\ &\qquad \Rightarrow Task_{(i,s)} \in RT_e^{DA})\end{aligned} \tag{10}$$

$$\begin{aligned}&\text{Exists } (Event_e.type = \text{"UnitFailure"}) \wedge\\ &\text{Exists } (Task_{(i,s)} \mid Task_{(i,s)} \in T_e^{IP} \wedge Task_{(i,s)}.assignedUnit = Event_e.unit),\\ &\quad \text{Let } Batch_i \leftarrow Task_{(i,s)}.batch,\\ &\qquad BatchTasks \leftarrow Batch_i.getTasks()\\ &\qquad \text{ForAll } (Task_{(i,s)} \mid Task_{(i,s)} \in BatchTasks \Rightarrow Task_{(i,s)} \in RT_e^{DA})\end{aligned} \tag{11}$$

When the plant operates under a non-intermediate storage, zero-wait (NIS-ZW) policy, the tasks that comprise a batch are highly coupled. Therefore, it is necessary to consider as members of the RT_e^{DA} set the following activities:

(i) All tasks that are associated with batches having an in-progress status at the *RTP* and that have, at least, one task assigned to the broken-down unit and such task is being executed at the *RTP* or is planned to start after it, but before the unit recovery time.
(ii) Tasks belonging to the T_e^{NE} set that are assigned to the broken-down unit and are associated with batches having a non-executed status, and which are planned to start later than the *RTP* but earlier than the unit recovery time.

The conditions that allow an automatic identification of these tasks are presented in expressions (12) and (13), respectively.

$$\begin{aligned}&\text{Exists}(Event_e.type="UnitFailure"),\\&\text{ForAll}(Batch_i \mid Batch_i.status="InProgress",\\&\text{Let } BatchTasks \leftarrow Batch_i.getTasks(),\\&\text{If Exists}\begin{pmatrix}Task_{(i,s)} \in BatchTasks \mid Task_{(i,s)}.assignedUnit = Event_e.unit \wedge\\((Task_{(i,s)}.actualStart \le Event_e.reschedtp \wedge\\ Task_{(i,s)}.plannedEnd \ge Event_e.reschedtp) \vee\\(Task_{(i,s)}.plannedStart \ge Event_e.reschedtp \wedge\\Task_{(i,s)}.plannedStart \le Event_e.unitRecoveryTime))\end{pmatrix}\\&\text{Then ForAll}(Task_{(i,s)} \mid Task_{(i,s)} \in BatchTasks \wedge Task_{(i,s)} \in T_e^{NE}\\&\qquad \Rightarrow Task_{(i,s)} \in RT_e^{DA}))\end{aligned} \tag{12}$$

$$\begin{aligned}&\text{Exists}(Event_e.type="UnitFailure"),\\&\text{ForAll}(Task_{(i,s)} \mid (Task_{(i,s)} \in T_e^{NE} \wedge Task_{(i,s)}.assignedUnit = Event_e.unit,\\&\text{Let } Batch_i \leftarrow Task_{(i,s)}.batch,\\&\text{If } (Batch_i.status="NonExecuted" \wedge\\&\qquad Event_e.reschedtp \le Task_{(i,s)}.plannedStart \wedge\\&\qquad Task_{(i,s)}.plannedStart \le Event_e.unitRecoveryTime)\\&\qquad \Rightarrow Task_{(i,s)} \in RT_e^{DA})\end{aligned} \tag{13}$$

The tasks that are identified as directly affected, independently of the interstage storage and operational policy, will be associated with a *ReassignAT* action type.

4.3.1.2. UBD event – indirectly affected tasks. When facing a unit breakdown event, the members of the RT_e^{IA} set are identified after specifying the members of the RT_e^{DA} set. This categorization process also takes into account the plant operating policy. If the plant operates under a UIS or a NIS-UW policy, the tasks that are considered to be indirectly affected by the event are the ones located downstream in those batches having a task that was identified as directly affected. This condition is captured by expression (14). Alternatively, if the workload of the system is high and there are few degrees of freedom to rearrange the agenda, the scheduler can expand this set by also including the tasks that are located upstream the directly affected one (see expression (15)).

$$\begin{aligned}&\text{Exists}(Event_e.type="UnitFailure"),\\&\text{ForAll}(Task_{(i,s)} \mid Task_{(i,s)} \in RT_e^{DA},\\&\qquad \text{Let } Batch_i \leftarrow Task_{(i,s)}.batch\\&\qquad\quad SuccessorSet \leftarrow Batch_i.getSuccessors(Task_{(i,s)})\\&\qquad\quad \text{ForAll}(Task_{(i,s)} \mid Task_{(i,s)} \in SuccessorSet \Rightarrow Task_{(i,s)} \in RT_e^{IA}))\end{aligned} \tag{14}$$

$$\begin{aligned}&\text{Exists}(Event_e.type="UnitFailure"),\\&\text{ForAll}(Task_{(i,s)} \mid Task_{(i,s)} \in RT_e^{DA},\\&\qquad \text{Let } Batch_i \leftarrow Task_{(i,s)}.batch\\&\qquad\quad BatchTasks \leftarrow Batch_i.getTasks()\\&\qquad\quad \text{ForAll}(Task_{(i',s)} \mid Task_{(i',s)} \in BatchTasks \wedge Task_{(i',s)} \ne Task_{(i,s)}\\&\qquad\qquad \Rightarrow Task_{(i',s)} \in RT_e^{IA}))\end{aligned} \tag{15}$$

On the other hand, if the plant operates under a NIS-ZW policy, the indirectly affected tasks are those concerning the batches that have a non-executed status at the *RTP* and have a directly affected task specified by means of expression (13). These indirectly affected activities are then captured by expression (16).

$$\begin{aligned}&\text{Exists}(Event_e.type="UnitFailure"),\\&\text{ForAll}(Task_{(i,s)} \mid (Task_{(i,s)} \in T_e^{DA}\\&\quad \text{Let } Batch_i \leftarrow Task_{(i,s)}.batch,\\&\quad \text{If } Batch_i.status="NonExecuted"\\&\qquad \text{Let } BatchTasks \leftarrow Batch_i.getTasks()\\&\qquad \text{ForAll}(Task_{(i',s)} \mid Task_{(i',s)} \in BatchTasks \wedge Task_{(i',s)} \ne Task_{(i,s)}\\&\qquad\qquad \Rightarrow Task_{(i',s)} \in RT_e^{IA}))\end{aligned} \tag{16}$$

4.3.1.3. UBD event – non-affected tasks. Independently of the intermediate storage and operation policy, this set of tasks includes the ones that are neither directly nor indirectly affected by the unit breakdown event, as specified by expression (17).

$$\begin{aligned}&\text{Exists}(Event_e.type = \text{“}UnitFailure\text{”}),\\&\text{ForAll}(Task_{(i,s)} \mid Task_{(i,s)} \in T_e^{NE} \wedge Task_{(i,s)} \notin RT_e^{DA} \wedge Task_{(i,s)} \notin RT_e^{IA})\\&\qquad \Rightarrow Task_{(i,s)} \in RT_e^{NA})\end{aligned} \tag{17}$$

In case a UBD event is faced, different action types and actions can be associated with tasks belonging to the RT_e^{IA} and RT_e^{NA} sets. As it will be shown when solving the case studies in Section 6, they can range from absolutely non-disrupting (*Freeze*), to very disruptive ones (*ReassignAT*), passing through a *Shift-JumpAT* that can be mild.

4.3.2. Rush batch arrival (RBA) event

When an instance of this type of event occurs, it contains information about the product to be produced and the recipe to be applied to manufacture the required batch, as well as the batch's deadline. This information is used in the rescheduling problem specification.

4.3.2.1. RBA event – directly affected tasks. Tasks needed to produce the batch to be inserted, which are members of the T_e^{NW} set (see expression (6)), are considered as directly affected (see expression (18)) and will be subject to an *AssignAT* action type.

$$\begin{aligned}&\text{Exists}(Event_e.type = \text{“}NewBatchArrival\text{”}),\\&\text{ForAll}(Task_{(i,s)} \mid Task_{(i,s)} \in T_e^{NW},\ Task_{(i,s)} \in RT_e^{DA})\end{aligned} \tag{18}$$

A similar approach can be followed in case the scheduler, due to quality problems, needs to associate additional processing tasks to a batch (see Fig. 5).

4.3.2.2. RBA event - Indirectly affected tasks. There are no indirectly affected tasks when a new batch arrival event takes place.

4.3.2.3. RBA event - Non-affected tasks. This set of tasks includes all the ones that were identified as non-executed yet at the time of the event, as indicated by expression (19).

$$\begin{aligned}&\text{Exists}(Event_e.type = \text{“}NewBatchArrival\text{”}),\\&\text{ForAll}(Task_{(i,s)} \mid Task_{(i,s)} \in T_e^{NE} \Rightarrow\ \ Task_{(i,s)} \in RT_e^{NA})\end{aligned} \tag{19}$$

4.3.3. Batch cancellation (BC) event

4.3.3.1. BC event – directly and indirectly affected tasks. When a BC event occurs, there are no directly affected tasks neither indirectly affected ones within the set of activities RT_e to be considered when tackling the resulting rescheduling problem. Since the set of tasks associated with the cancelled batch are part of the NT_e set (see expression (9)), they will be automatically eliminated from the problem specification.

4.3.3.2. BC event – non-affected tasks. This set of tasks includes all the tasks that were identified as non-executed yet at the time of the batch cancellation event, as indicated by expression (20).

$$\begin{aligned}&\text{Exists}\,(Event_e.type = \text{“}BatchCancellation\text{”}),\\ &\quad\text{ForAll}\,(Task_{(i,s)}\,|Task_{(i,s)} \in \mathrm{T}_e^{NE} \wedge Task_{(i,s)} \notin \mathrm{T}_e^{CA}\\ &\qquad\Rightarrow\ Task_{(i,s)} \in \mathrm{RT}_e^{NA})\end{aligned} \tag{20}$$

4.4. Alternative scenarios for indirectly-affected and non-affected tasks

Both indirectly and non-affected tasks are the ones that are supposed to be less influenced by disrupting events. If maintaining schedule stability is a goal, a large fraction of the scheduling decisions associated with these tasks should remain the same, or at most experience limited modifications during the rescheduling process. In consequence, tasks belonging to these categories are not to be associated with unsettling actions or action types.

In general, it is desired not to move/reassign those non-affected tasks having a start time which is close to the rescheduling time point. As it was pointed out by Musier and Evans (1990), whenever possible, batches which are about to be processed should not be moved to another unit or to another position in the sequence because their required raw materials have already been sent. Based on these ideas Méndez and Cerdá (2003) proposed freezing a portion of the schedule, without giving hints on which tasks to immobilize. In this contribution, we suggest as a possible scenario to freeze those tasks that belong to the RT_e^{NA} set and have a planned start time close to the rescheduling time point. Thus, as shown in Fig. 7 a freezing period *FP* that only applies for tasks in RT_e^{NA} can be defined. The challenge is to reasonably set the limits of such period, which will define which are the non-affected tasks to be frozen. The larger the *FP* period, the more stable the newly generated schedule becomes, but probably less effective it turns out to be in terms of regular performance measures. In general, the length of the *FP* interval depends on both, (i) the average processing time of the tasks to be scheduled, and (ii) the ratio of this average processing time to the length of the remaining portion of the scheduling horizon. Thus, the *FP* length, which is problem specific, is inversely proportional to the value of these two problem elements. In fact, shops having tasks with large average processing time values are less prone to nervousness; therefore, the freezing interval does not need to be large. Similarly, if the above-mentioned ratio tends to one, there is almost no flexibility to accommodate the schedule; so, the tasks to be frozen have to be reduced to a minimum (the length of the freezing period should be minimized).

For tasks that are not frozen, the less stirring action type to be applied to them is the *Shift-JumpAT* one. This action type can be applied to tasks belonging to sets RT_e^{NA} and RT_e^{IA}, having planned start times located in the neighbourhood of the *RTP* (within the *FP*) and, more likely, within a period located a little bit beyond it, referred as the *S-JP* period (see Fig. 7). Finally, tasks in RT_e^{NA} and RT_e^{IA} having planned start times located further than the *FP* and *S-JP* intervals can be subject to different types of rescheduling actions, including more disrupting ones like *ReassignAT*, giving rise to a reassign period *RP*. In consequence, both indirectly and non-affected tasks of a certain rescheduling problem, can be associated with different rescheduling action types or actions, based on their context. So, alternative solution scenarios can be generated for each particular problem depending on the status and intrinsic characteristics of the scheduling environment and the scheduler's criteria. Each scenario will be characterized by (i) the length of its associated *FP* and *S-JP* intervals and (ii) the types of actions to be applied to those tasks having planned start times located within these intervals. The lengths of the *FP* and *S-JP* periods, which are problem dependent, will be specified as a multiple of the average processing time of all the tasks involved in the rescheduling problem. In the calculation of this overall parameter the processing time of each task is computed as an average value of the processing times in all the equipment items where the task can be executed.

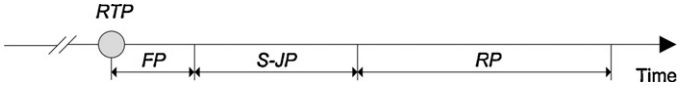

Fig. 7. Time periods within the rescheduling horizon.

Alternative solution scenarios can be also generated by assigning different values to the *ls* and *rs* parameters of the *Shift-JumpAT*(⟨*Task*⟩, *ls*, *rs*) action type. The maximum value that can be assigned to the *ls* parameter adopted for $Task_{(i,s)}$ (named $lsmax_{(i,s)}$) is given by expression (21), where $est_{(i,s)}$ stands for the earliest start time of $Task_{(i,s)}$, rd_u represents the release time of the unit where such task is assigned, and RT_e^S stands for the set of tasks in RT_e for which a shift-jump action type has been decided.

$$\begin{aligned}&lsmax_{(i,s)} = Task_{(i,s)}.plannedStart - Max\ \ (reschedtp, rd_u, est_{(i,s)});\\ &\forall\ \ Task_{(i,s)} \in \mathrm{RT}_e^S, Task_{(i,s)}.assignedUnit = u\end{aligned} \tag{21}$$

On the other hand, the maximum value that can be assigned to the *rs* parameter that is adopted for $Task_{(i,s)}$ (named $rsmax_{(i,s)}$) is given by expression (22).

$$\begin{aligned}&rsmax_{(i,s)} = scheduleHorizon - Task_{(i,s)}.plannedEnd;\\ &\forall\ \ Task_{(i,s)} \in \mathrm{RT}_e^S\end{aligned} \tag{22}$$

If the schedule stability is tried to be preserved, values of $ls_{(i,s)}$ and $rs_{(i,s)}$ lower than $lsmax_{(i,s)}$ and $rsmax_{(i,s)}$ are to be selected. Expressions (23) and (24) provide general expressions for the values to be assigned to these parameters, which are directly proportional to the task duration.

$$\begin{aligned}&ls_{(i,s)} = \alpha_{i,s} Task_{(i,s)}.duration \wedge ls_{(i,s)} \le lsmax_{(i,s)};\\ &\forall\ \ Task_{(i,s)} \in \mathrm{RT}_e^S\end{aligned} \tag{23}$$

$$\begin{aligned}&rs_{(i,s)} = \beta_{i,s} Task_{(i,s)}.duration \wedge rs_{(i,s)} \le rsmax_{(i,s)};\\ &\forall\ \ Task_{(i,s)} \in \mathrm{RT}_e^S\end{aligned} \tag{24}$$

Thus, the values of $ls_{(i,s)}$ and $rs_{(i,s)}$ can be adjusted by modifying the values of the $\alpha_{(i,s)}$ and $\beta_{(i,s)}$ parameters. Assuming that all the tasks assigned to a given unit have similar processing times, if $\alpha_{(i,s)}$ and $\beta_{(i,s)}$ are given positive values lower than one, the task will only be allowed to make shift movements, just sliding to the left or to the right. As the values of $\alpha_{(i,s)}$ and $\beta_{(i,s)}$ increase over the unitary value, the task is given more flexibility and is allowed modifying its current sequence position and doing some "jumping" to reach a different one. As mentioned before, the decision of assigning greater values to these parameters and giving more flexibility to this action type depends on the scenario, the type of task (indirectly-affected or non-affected), the workload of the assigned unit, etc.

5. Model generation

Once the rescheduling problem has been properly specified and the performance measure to be used has been selected, the model generation module is in charge of setting up the corresponding constraint programming (CP) model. The CP approach has been chosen because of the highly declarative nature of constraint-based languages, the possibilities of incorporating new constraints incrementally as well as developing/adopting search strategies that take advantage of the specific characteristics of the problem at hand.

Other advantages are the potential to detect unfeasible problem specifications immediately, to find initial feasible solutions quite fast and to obtain optimal and suboptimal solutions in reduced CPU times. This last feature is especially appealing in the context of rescheduling problems that demand immediate responses to the occurrence of unexpected events.

To implement the CP models, the OPL language, which is the underlying language of the ILOG OPL Studio environment (ILOG, 2002), along with the ILOG Scheduler package (ILOG, 2000) have been selected. This last package employs some specific scheduling constructs, such as (i) *requires*, which enforces the assignment of a renewable resource demanded by an activity, from a set of alternative resources (ii) *precedes*, that does not allow tasks belonging to the same batch to overlap, (iii) *activityHasSelectedResource*, which acts as a predicate that evaluates to true when a task has been assigned to a resource belonging to a set of alternative resources, and (iv) *break*, which is a construct that specifies a period of unavailability on a given renewable resource. Furthermore, this ILOG environment allows implementing a search strategy by employing domain knowledge. The chosen strategy depends on the rescheduling scenario that is being tackled. In the examples presented in this paper, a search strategy based on the equipment workload balance, which was proposed and assessed by Zeballos, Novas and Henning (2010), has been adopted.

5.1. Constraint programming model

In order to develop the CP model corresponding to a given solution scenario, it is necessary to translate the actions types associated with it into constraints. The set of assignment, timing and/or precedence constraints includes event-dependent ones, as well as others that are characteristic of any batch scheduling problem, which are referred as basic constraints.

The construction of the CP model also implies that for each available processing unit it is necessary to establish its ready time. It is calculated by taking the maximum value of *RTP* and the planned end of the in-progress task assigned to it, if any. In case that $Event_e$ is associated with a unit failure, the ready time will be equal to the unit recovery time. Similarly, for each task in RT_e its release time needs to be calculated/estimated. For tasks having a direct predecessor in the T_e^{AE} set, their release time will be equal to the rescheduling time point. For those having a direct predecessor in the T_e^{IP} set, their release time will equal to the planned end of their predecessor task. In the case of tasks having a direct predecessor that does not belong to sets T_e^{AE} or T_e^{IP}, their release time will be estimated as the batch release time plus the summation of the minimum processing times of their predecessors in the batch sequence. Finally, the release time of those tasks which are the first in the batch sequence, is equal to the batch release time.

5.1.1. Basic constraints

Basic constraints are generated for all the tasks in the RT_e set. Expression (25) is an assignment constraint prescribing that each processing task must be allocated to just one unit belonging to the set of alternative units where it can be processed.

$$Task_{(i,s)} \;\; requires \;\; U_{(i,s)}, \quad \forall \; Task_{(i,s)} \in RT_e, \quad \forall i \in I, \quad \forall s \in S \tag{25}$$

Constraint (26) enforces the proper sequencing of all the processing operations corresponding to consecutive stages (s and s') that need to be executed on each batch i.

$$Task_{(i,s)} \;\; precedes \;\; Task_{(i,s')} \quad \forall \; Task_{(i,s)}, Task_{(i',s')} \in RT_e, \quad \forall i, i' \in I, \quad \forall s, s' \in S, s \neq last(S), Ord(s') = Ord(s) + 1 \tag{26}$$

If the intermediate storage policy is assumed to be UIS, constraint (26) is enough to establish the timing of the activities associated with a batch. However, to handle a NIS/ZW policy the constraint (27) is to be included in the model. It enforces the end of a processing activity required by a batch to coincide with the beginning of the next operation (no waiting between consecutive stages)

$$Task_{(i,s)}.end = Task_{(i,s')}.start \quad \forall \; Task_{(i,s)}, Task_{(i',s')} \in RT_e, \quad \forall i, i' \in I, \quad \forall s, s' \in S, s \neq last(S), Ord(s') = Ord(s) + 1 \tag{27}$$

Expression (28) places a lower bound on the start time of tasks based on the ready time of the processing unit where each task is assigned and on the release time of the task.

$$ActivityHasSelectedResource \;\; (Task_{(i,s)}, U_{(i,s)}, u) \Rightarrow Task_{(i,s)}.start > Max(rd_u, rt_{(i,s)}) \quad \forall \; Task_{(i,s)} \in RT_e, \quad \forall u \in U_{(i,s)} \tag{28}$$

Expression (29) models a topological constraint, imposing that two consecutive tasks of a certain batch are not to be assigned to unconnected units.

$$ActivityHasSelectedResource(Task_{(i,s)}, U_{(i,s)}, u) \Rightarrow not\, ActivityHasSelectedResource(Task_{(i,s')}, U_{(i,s')}, u') \quad \forall \; Task_{(i,s)} \in RT_e, \quad \forall s, s' \in S, s \neq last(S), Ord(s') = Ord(s) + 1, \quad \forall u, u' \in U, u' \notin CU_u \tag{29}$$

5.1.2. Event-dependent constraints

Constraints of this type depend on the disrupting situation being faced and on the actions and action types adopted for the tasks included in sets RT_e^{DA}, RT_e^{IA}, and RT_e^{NA}. The association between a task and a rescheduling action type, developed in the rescheduling problem specification, establishes the constraints that are required by such task to be properly rescheduled.

5.1.2.1. Shift-jump action types on tasks. Tasks belonging to sets RT_e^{IA} and RT_e^{NA} for which a shift-jump action type has been decided are included in the set RT_e^S. Constraint (30) enforces these tasks to maintain their current unit assignments and their start times within the allowed limits, whereas constraints (31) and (32) set the task durations, which depend on the adopted interstage policy. Expression (31) holds for UIS and NIS-ZW, and (32) for NIS-UW. In these expressions, u represents the unit allocated to $Task_{(i,s)}$ in the active agenda (the value of $Task_{(i,s)}.assignedUnit$) and parameters $ls_{(i,s)}$ and $rs_{(i,s)}$ stand for the left and right shift limits corresponding to the *Shift-JumpAT* action type associated with $Task_{(i,s)}$.

$$\begin{aligned} &ActivityHasSelectedResource(Task_{(i,s)}, U_{(i,s)}, u) \wedge \\ &not\, ActivityHasSelectedResource(Task_{(i,s)}, U_{(i,s)}, u') \Rightarrow \\ &Task_{(i,s)}.start \leq Task_{(i,s)}.plannedStart + rs_{(i,s)} \wedge \\ &Task_{(i,s)}.start \geq Task_{(i,s)}.plannedStart - ls_{(i,s)} \\ \forall \; &Task_{(i,s)} \in RT_e^S, \quad \forall u' \in U_{(i,s)}, u = Task_{(i,s)}.assignedUnit, u' \neq u \end{aligned} \tag{30}$$

$$ActivityHasSelectedResource(Task_{(i,s)}, U_{(i,s)}, u) \Rightarrow Task_{(i,s)}.duration = Task_{(i,s)}.plannedDuration, \quad \forall \; Task_{(i,s)} \in RT_e^S, \quad \forall u \in U_{(i,s)} \tag{31}$$

$$\begin{aligned}&ActivityHasSelectedResource(Task_{(i,s)}, U_{(i,s)}, u)\\&\quad\Rightarrow Task_{(i,s)}.duration \geq pt_{i,s,u},\\&\quad\forall\ \ Task_{(i,s)} \in RT_e^S,\quad \forall s \in S, s \neq last(S),\quad \forall u \in U_{(i,s)}\\&ActivityHasSelectedResource(Task_{(i,s)}, U_{(i,s)}, u)\\&\quad\Rightarrow Task_{(i,s)}.duration = pt_{i,s,u},\\&\quad\forall\ \ Task_{(i,s)} \in RT_e^S,\quad \forall s \in S, s = last(S),\quad \forall u \in U_{(i,s)}\end{aligned} \tag{32}$$

where $pt_{i,s,u}$ represents the processing time demanded by $Task_{(i,s)}$ when it is executed on $Unit_u$.

5.1.2.2. Assign and reassign action types on tasks. Tasks belonging to sets RT_e^{DA}, RT_e^{IA} and RT_e^{NA} for which an assign or reassign action type have been chosen are included in the set RT_e^A. Constraint (33) prescribes that each task $Task_{(i,s)}$ belonging to the set RT_e^A, requires being assigned to any available equipment item which can process such task and is available during the rescheduling period (a unit belonging to the set $U_{(i,s)}$). Also, the constraint prescribes that the task duration depends on the assigned unit, since $pt_{i,s,u}$ represents the processing time demanded by $Task_{(i,s)}$ when it is executed on $Unit_u$. It should be noted that the set $U_{(i,s)}$ can include the equipment item where $Task_{(i,s)}$ is already assigned.

$$\begin{aligned}&ActivityHasSelectedResource(Task_{(i,s)}, U_{(i,s)}, u) \Rightarrow\\&\qquad Task_{(i,s)}.duration = pt_{i,s,u},\\&\quad\forall\ \ Task_{(i,s)} \in RT_e^A,\quad \forall u \in U_{(i,s)}\end{aligned} \tag{33}$$

In case the NIS/UW policy rules the interstage handling of materials, constraint (33) should not be included in the model. Instead, constraints (34) and (35) must replace it.

$$\begin{aligned}&ActivityHasSelectedResource(Task_{(i,s)}, U_{(i,s)}, u) \Rightarrow\\&\qquad Task_{(i,s)}.duration \geq pt_{i,s,u},\\&\forall\ \ Task_{(i,s)} \in RT_e^A,\quad \forall s \in S, s \neq last(S),\quad \forall u \in U_{(i,s)}\end{aligned} \tag{34}$$

$$\begin{aligned}&ActivityHasSelectedResource(Task_{(i,s)}, U_{(i,s)}, u) \Rightarrow\\&\qquad Task_{(i,s)}.duration = pt_{i,s,u},\\&\quad\forall\ \ Task_{(i,s)} \in RT_e^A, s = last(S),\quad \forall u \in U_{(i,s)}\end{aligned} \tag{35}$$

5.1.2.3. Freeze action types. Tasks in RT_e^{NA}, having a planned start time within the *FP* period that are required to maintain their unit allocations and their current start times, are included in the set RT_e^F. They are forced to be frozen by means of constraint (36), which is applicable to any task, and expressions (37) or (38), depending on the adopted interstage policy. Expression (37) holds for UIS or NIS-ZW policies, whereas (38) is applicable to NIS-UW situations. In these expressions, u represents the unit already allocated to $Task_{(i,s)}$ in the agenda that was active prior to the unexpected event occurrence.

$$\begin{aligned}&ActivityHasSelectedResource(Task_{(i,s)}, U_{(i,s)}, u)\ \ \wedge\\&not\ ActivityHasSelectedResource(Task_{(i,s)}, U_{(i,s)}, u') \Rightarrow\\&Task_{(i,s)}.start = Task_{(i,s)}.plannedStart\\&\quad\forall\ \ Task_{(i,s)} \in RT_e^F,\quad \forall u' \in U_{(i,s)}, u = Task_{(i,s)}.assignedUnit, u' \neq u\end{aligned} \tag{36}$$

$$\begin{aligned}&ActivityHasSelectedResource(Task_{(i,s)}, U_{(i,s)}, u) \Rightarrow\\&Task_{(i,s)}.duration = Task_{(i,s)}.plannedDuration;\\&\quad\forall\ \ Task_{(i,s)} \in RT_e^S,\quad \forall u \in U_{(i,s)}\end{aligned} \tag{37}$$

$$\begin{aligned}&ActivityHasSelectedResource(Task_{(i,s)}, U_{(i,s)}, u) \Rightarrow\\&\qquad Task_{(i,s)}.duration \geq pt_{i,s,u},\\&\forall\ \ Task_{(i,s)} \in RT_e^S,\quad \forall s \in S, s \neq last(S),\quad \forall u \in U_{(i,s)}\\&ActivityHasSelectedResource(Task_{(i,s)}, U_{(i,s)}, u) \Rightarrow\\&\qquad Task_{(i,s)}.duration = pt_{i,s,u},\\&\forall\ \ Task_{(i,s)} \in RT_e^S,\quad \forall s \in S, s = last(S),\quad \forall u \in U_{(i,s)}\end{aligned} \tag{38}$$

5.1.2.4. Renewable resource failure. Expression (39) specifies an unavailability period for a unit that failed. It occurs when $Event_e$ corresponds to a unit failure that extends from *RTP* to the recovery time of the unit. The same expression can be applied to any other critical renewable resource.

$$\begin{aligned}&Break(u, Event_e.reschedtp, Event_e.unitRecoveryTime),\\&Event_e.type = \text{“}UnitFailure\text{”}, Event_e.unit = u\end{aligned} \tag{39}$$

5.1.3. Performance measures

When addressing a rescheduling problem, as pointed out in Section 2.3, in addition to classical performance measures used to assess the efficiency of the agenda (e.g., makespan, tardiness, earliness, etc.), stability measures associated with a smooth operation of the plant need to be taken into account.

In this contribution, the case-studies that are presented in Section 6 have been solved employing the minimization of Makespan (*Mk*) and Total Deviation (*TD*) as the objective functions. When minimizing one of these performance measures, the impact over the other one has also been evaluated. Furthermore, other metrics, such as the Total Completion Time (*TCT*), Normalized Equipment Stability (*NES*) and Number of Temporal Shifted Tasks (*NST*) have been calculated with the aim of having more information about the overall quality of the newly generated schedules.

5.1.3.1. Makespan. This measure represents the maximum completion time among all the batches that participate in the rescheduling problem. When the minimization of the *Mk* variable is pursued, expression (40) has to be included in the model.

$$Task_{(i,s)}\ \ precedes\ Mk,\quad \forall\ \ Task_{(i,s)} \in RT_e, s = last(S) \tag{40}$$

5.1.3.2. Total deviation. This measure quantifies the changes over the planned start times of all the tasks included in RT_e. If the minimization of *TD* is chosen as the objective function, expression (41) has to be included in the CP model.

$$TD = \sum_{Task_{(i,s)} \in RT_e} \left| Task_{(i,s)}.start - Task_{(i,s)}.plannedStart \right| \tag{41}$$

5.1.3.3. Additional metrics. One of the performance measures that allows assessing the quality of a schedule is *TCT*, which represents the summation of the completion times of all the batches that participate in the rescheduling problem (see expression (42)).

$$TCT = \sum_{Task_{(i,s)} \in RT_e : s = last(S)} Task_{(i,s)}.end \tag{42}$$

Another regular evaluation metric that can be employed to assess schedules when due dates are relevant is *TT*, which represents the total tardiness. As expression (43) prescribes, *TT* is calculated as the sum of the delays in the batch completion times with respect to their due dates.

$$TT = \sum_{Task_{(i,s)} \in RT_e : s = last(S)} Max\left(0, Task_{(i,s)}.end - dd_i\right) \tag{43}$$

where dd_i stands for $Batch_i.dueDate$. Similarly, performance measures related to earliness or just-in-time manufacture can also be proposed and chosen by the scheduler.

In addition to the previous regular measures, schedule stability appraisals other than *TD* are needed. Expression (44) introduces *NES*, a normalized equipment stability measure that assumes a value equal to one if no task has changed its originally assigned unit and zero if all the tasks have changed their allocated

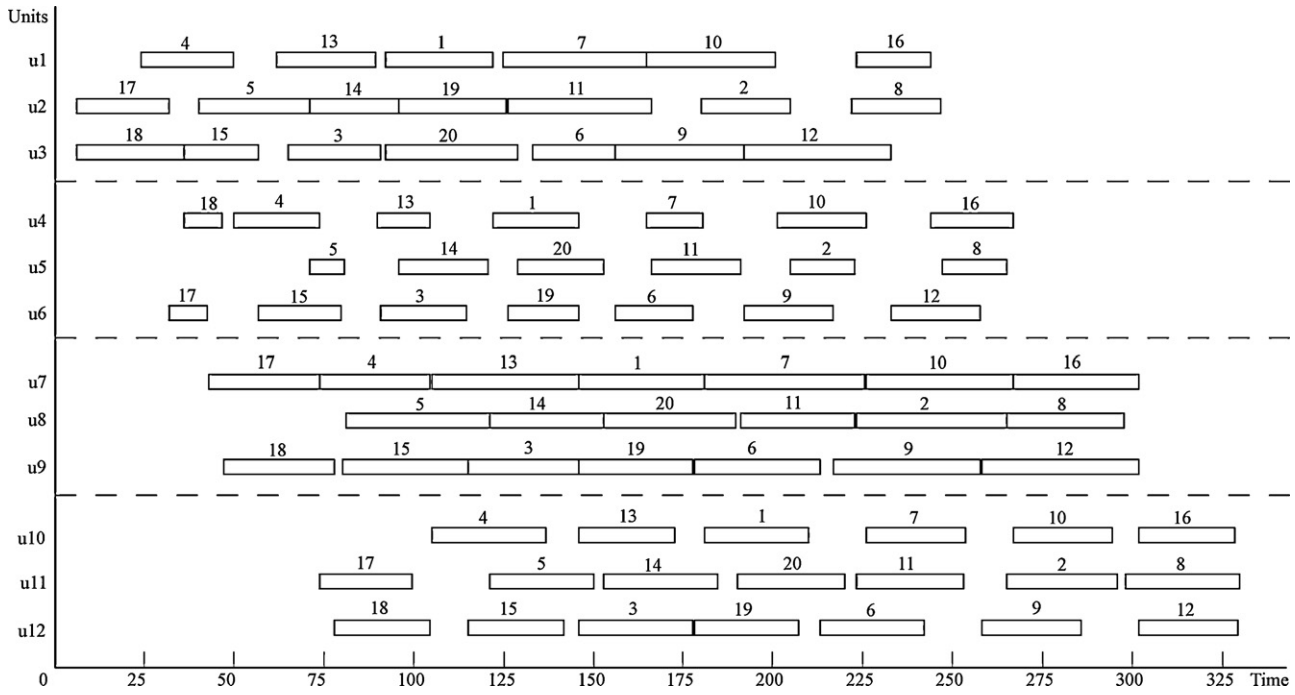

Fig. 8. In-progress schedule, NIS-ZW policy.

equipment item.

$$NES = 1 - \frac{\sum_{Task_{(i,s)} \in \mathrm{RT}_e} Y_{(i,s)}}{Card(\mathrm{RT}_e)}$$

where

$$Y_{(i,s)} \begin{pmatrix} 1 \leftarrow \text{if } ActivityHasSelectedResource\,(Task_{(i,s)}, \mathrm{U}_{(i,s),}, u) = 1 \wedge \\ \quad Task_{(i,s)}.assignedUnit \neq u \wedge u' \neq u \\ 0 \leftarrow \text{if } ActivityHasSelectedResource(Task_{(i,s)}, \mathrm{U}_{(i,s),}, u) = 1 \wedge \\ \quad Task_{(i,s)}.assignedUnit = u \end{pmatrix} \tag{44}$$

Another stability measure is defined by means of expression (45), which introduces an assessment normalized measure named *NST*, which quantifies the number of tasks in $\mathrm{RT_e}$ that have varied their planned start times, independently if they have changed their originally assigned unit or not. *NST* assumes a value equal to one if no task in RT_e has changed its original planned start time and zero in case all the tasks have altered their initial times.

$$NST = 1 - \frac{\sum_{Task_{(i,s)} \in \mathrm{RT}_e} X_{(i,s)}}{Card(\mathrm{RT}_e)}$$

where (45)

$$X_{(i,s)} \begin{pmatrix} 1 \leftarrow \text{if } Task_{(i,s)}.start \neq Task_{(i,s)}.plannedStart \\ 0 \leftarrow \text{if } Task_{(i,s)}.start = Task_{(i,s)}.plannedStart \end{pmatrix}$$

6. Examples and computational results

With the aim of testing this proposal, various examples have been solved and some of them are reported in this section. They are based on a case-study first introduced by Castro and Grossmann (2005) as problem P10, which has been modified to enlarge it. The multiproduct batch plant involves 4 stages and 12 units, instead of 8 as in the original problem; so at each stage one of the equipment items is duplicated. The plant is assumed to operate under a NIS-ZW interstage storage policy. During the scheduling horizon, 20 batches (2 per each order of the P10 problem) have to be processed, minimizing the total amount of time it takes to complete all jobs. Fig. 8 shows the in-progress schedule that has a Makespan of 330 time units. Provided that unit failure is one of the most disrupting events that can occur, different examples of equipment breakdown are considered.

Table 1
Data for different equipment failure cases.

Case	Affected unit	Rescheduling time point	Recovery time
Case 1	3	45	145
Case 2	3	120	220
Case 3	7	45	145
Case 4	7	150	250

The various cases of equipment failure are shown in Table 1. As seen, unit breakdowns are assumed to occur in units u3 and u7, thus affecting stages 1 and 3, this last one being the bottleneck of the production environment. Besides testing the effect of a unit failure at different stages, the examples allow investigating the effect of a unit failure when it takes place at different time points of the scheduling horizon. In order to make a fair comparison it will be assumed that in all cases it takes 100 time units to have the equipment item repaired and operable again.

Following the proposed solution methodology, tasks belonging to each case study were automatically classified according to the proposal presented in Section 4.1 that takes into account the status of the plant, as well as the impact of the unexpected event. Thus, sets RT_e^{DA}, RT_e^{IA} and RT_e^{NA} were automatically identified for all the case studies. Table 2 shows the number of tasks comprising each of these sets for the various examples. In all the situations it has been assumed that the batch that is being manufactured when the unit breakdown occurs has to be redone. Fig. 9 depicts the tasks comprising each of the sets for Case 3, showing that batch 17 is the one that has to be manufactured again.

Having done this task classification, several solution scenarios for the equipment breakdown event can be proposed based on the adoption of different action types for tasks that are non-affected or indirectly affected by the disruptive event. Therefore,

Table 2
Classification of tasks for each case study.

Sets	Case study			
	1	2	3	4
RT_e^{DA}	7	7	9	9
RT_e^{IA}	9	9	3	3
RT_e^{NA}	57	35	64	30
RT_e	73	51	76	42

each one of the four cases studies described in Tables 1 and 2 will be addressed under six different solution scenarios. Table 3 summarizes the rescheduling action types associated with sets RT_e^{DA}, RT_e^{IA} and RT_e^{NA} for the six scenarios. Since the unexpected event type is a unit failure, all the tasks that were directly affected will be reassigned. In the case of the scenarios 5 and 6 tasks belonging to the RT_e^{NA} set were associated with different action types according to their location with respect to the rescheduling time point. Therefore, for these two scenarios the position of the non-affected tasks in relation to the *FP* and *S-JP* periods determined the action type to be applied to them, being frozen the ones having a planned start time located within the *FP* period and being associated with a *Shift-JumpAT* those with a planned start time situated within the *S-JP* period. Tasks having a planned start time located beyond this last period were allowed to be reassigned. For the examples associated with scenarios 5 and 6, the length of *FP* period was adopted to be equal to one and two times (Freeze factor equal to 1 and 2) the average processing time of the tasks involved in the problem, respectively. Similarly, the length of the *S-JP* period was assumed to be equal to three and six times (Shift-Jump factor equal to 3 and 6) this average processing time.

The four case-studies were tackled under the six scenarios described in Table 3 and each problem instance was solved with two objective functions. Tasks for which a *Shift-JumpAT* action type was chosen were associated with $lsmax_{(i,s)}$ and $\beta_{i,s} = 5$ values, with exception of those in which this $\beta_{i,s}$ value led to infeasibilities. In such cases, bigger values of $\beta_{i,s}$ had to be adopted. The CP models corresponding to all the combinations of case studies and scenarios were automatically generated based on their specifications. Instead of the ILOG default search process (Depth First Search, DFS) a search strategy that pursues a balance of the equipment workload was implemented and used to solve all the problem instances, since it was proved (Zeballos et al., 2010) that in most of the examples such strategy outperforms DFS. Examples were solved with a PC having a Pentium Dual Core 3.4 GHz processor with 2 GB of RAM. Makespan (*Mk*) was chosen as the regular performance to optimize and Total Deviation (*TD*) was selected as the measure of schedule instability to be minimized.

Table 4 presents the results reached when minimizing Makespan. In turn, Table 5 shows a comparison of the best solutions

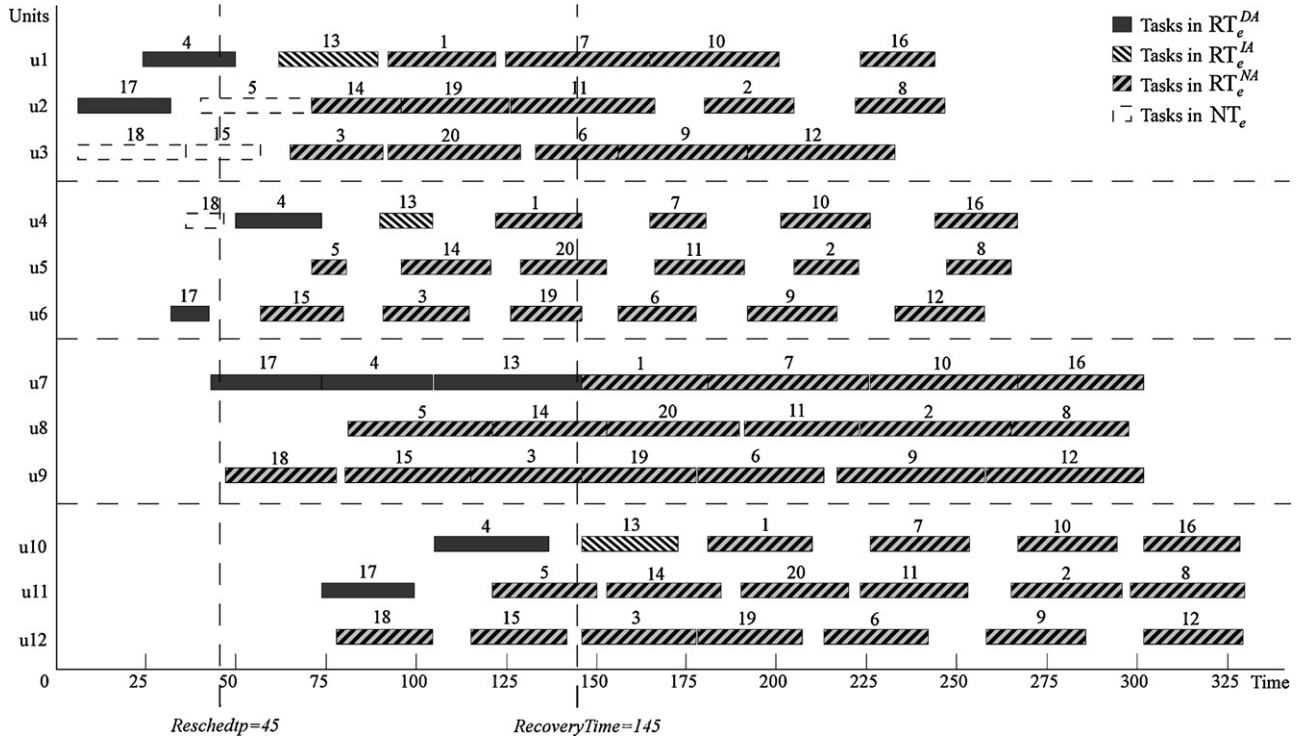

Fig. 9. Directly, indirectly and non-affected tasks for Case Study 3.

Table 3
Different solution scenarios for a unit failure event.

Alternative scenarios	Tasks in RT_t^{DA}	Tasks in RT_t^{IA}	Tasks in RT_t^{NA}
Scenario 1	*ReassignAT*	*Shift-JumpAT*	*FreezeAT*
Scenario 2	*ReassignAT*	*Shift-JumpAT*	*Shift-JumpAT*
Scenario 3	*ReassignAT*	*ReassignAT*	*Shift-JumpAT*
Scenario 4	*ReassignAT*	*ReassignAT*	*ReassignAT*
Scenario 5 & Scenario 6	*ReassignAT*	*ReassignAT*	Based on *FP, S-JP & RP* periods

Scenario 5: Freeze factor = 1; Shift-Jumpt factor = 3.
Scenario 6: Freeze factor = 2; Shift-Jumpt factor = 6

Table 4
Solutions reached when minimizing Makespan. NIS-ZW policy and $\beta_{i,s} = 5$.

Problem	Optimal/best solution in 300 s		Other performance measures			
	Mk	CPU time[a]	*TD*	*TCT*	*NST*	*NES*
Case 1						
Scen1	397[b]	<1	2855	5134	0.79	0.94
Scen2	356	<1	4264	4734	0.13	0.96
Scen3	355[c]	111.9	4531	4728	0.13	0.94
Scen4	371[c]	<1	3984	5025	0.24	0.92
Scen5	375[c]	177.7	2611	4931	0.29	0.94
Scen6	366[c]	240.1	2109	4945	0.40	0.87
Case 2						
Scen1	413	<1	1735	4367	0.69	0.97
Scen2	358	<1	1675	4122	0.38	0.95
Scen3	353	42.9	1886	4098	0.22	0.89
Scen4	353[c]	46.5	2190	4128	0.22	0.71
Scen5	361[c]	2.4	1732	4233	0.53	0.79
Scen6	361	3.3	1162	4196	0.46	0.87
Case 3						
Scen1	361[d]	<1	2730	5090	0.85	0.91
Scen2	364	4.5	3866	5058	0.11	0.91
Scen3	359	6.6	3294	5100	0.64	0.93
Scen4	382[c]	267.3	5216	5247	0.11	0.85
Scen5	359[c]	124.8	2701	5084	0.64	0.93
Scen6	360	8.5	2674	5089	0.79	0.93
Case 4						
Scen1	374	<1	1413	3779	0.72	0.81
Scen2	366	<1	1289	3750	0.62	0.81
Scen3	366	2.8	1380	3741	0.43	0.89
Scen4	360[c]	37.6	1913	3706	0.24	0.81
Scen5	364	15.5	1233	3737	0.43	0.89
Scen6	366	<1	1431	3752	0.53	0.84

[a] Time required to reach optimal solutions or to instantiate suboptimal ones.
[b] $\beta_{i,s} = 8$. Lower values of $\beta_{i,s}$ result in unfeasible solutions.
[c] Suboptimal solution.
[d] $\beta_{i,s} = 13$. Lower values of $\beta_{i,s}$ result in unfeasible solutions.

Table 5
Trade-offs between performance measures. NIS-ZW policy and $\beta_{i,s} = 5$. Minimization of Makespan and Total Deviation.

Problem	Objective function	Performance measures values					CPU time[a]
		Mk	*TD*	*TCT*	*NST*	*NES*	
Case 1—Scen3	*Mk*	**355**[b]	4531	4728	0.13	0.94	111.9
	TD	407	**2271**[b]	4988	0.50	0.99	46.3
Case 2—Scen3	*Mk*	**353**	1886	4098	0.22	0.89	42.9
	TD	381	**1201**[b]	4218	0.46	0.95	31.3
Case 3—Scen3	*Mk*	**359**	3294	5100	0.64	0.93	6.2
	TD	404	**2640**[b]	5088	0.43	0.97	252.7
Case 4—Scen4	*Mk*	**360**[b]	1913	3706	0.24	0.81	37.6
	TD	399	**1381**[b]	3791	0.62	0.89	106.6

[a] Time required to reach optimal Solutions or to instantiate suboptimal ones.
[b] Suboptimal solution.

obtained for this performance measure with the ones reached when minimizing total deviation (*TD*). Due to the fact that quick answers are mandatory when facing rescheduling problems, 300 s was the limit imposed to obtain solutions. In order to allow a comprehensive comparison of the various solutions obtained for each problem instance, other performance indexes were calculated. The other assessment measures were the Total Completion Time (*TCT*), as well as the normalized stability measures *NST* and *NES*, introduced by expressions (44) and (45).

Table 4 shows that optimal solutions were reached for 15 out of 24 problem instances in very low CPU times. For this production environment the number of suboptimal solutions diminished to just one when the number of batches was reduced to 10. Table 4 also shows that the computational effort is greater when

Table 6
Trade-offs between performance measures. NIS-UW policy and $\beta_{i,s} = 5$. Minimization of Makespan and Total Deviation.

Problem	Objective function	Performance measures values					CPU time[a]
		Mk	*TD*	*TCT*	*NST*	*NES*	
Case 1—Scen3	*Mk*	**347**	5547	4713	0.13	0.98	103.3
	TD	435	**2687**	5091	0.62	1.00	<1
Case 3—Scen3	*Mk*	**359**	3639	4986	0.16	0.94	3.1
	TD	435	**3715**	5340	0.47	0.98	188.7

[a] Time required to reach optimal solutions or to instantiate suboptimal ones
*Suboptimal solution

the event takes place at the beginning of the scheduling horizon. In addition, results indicate that the earlier the failure takes place, the more degrees of freedom the problem has and the wider the variety of solutions that can be obtained with the various scenarios. Therefore, when the equipment breakdown occurs at the beginning of the scheduling horizon it is worth to solve all the proposed scenarios or even additional ones and let the scheduler select the best solution according to his/her preferences.

An analysis of the results presented in Tables 4 and 5 confirms what can be intuitively predicted. In general, solutions corresponding to the minimization of regular performance measures, like Makespan, do not exhibit good stability indexes. On the other hand, when *TD* is minimized, the values of *Mk* noticeably degrade.

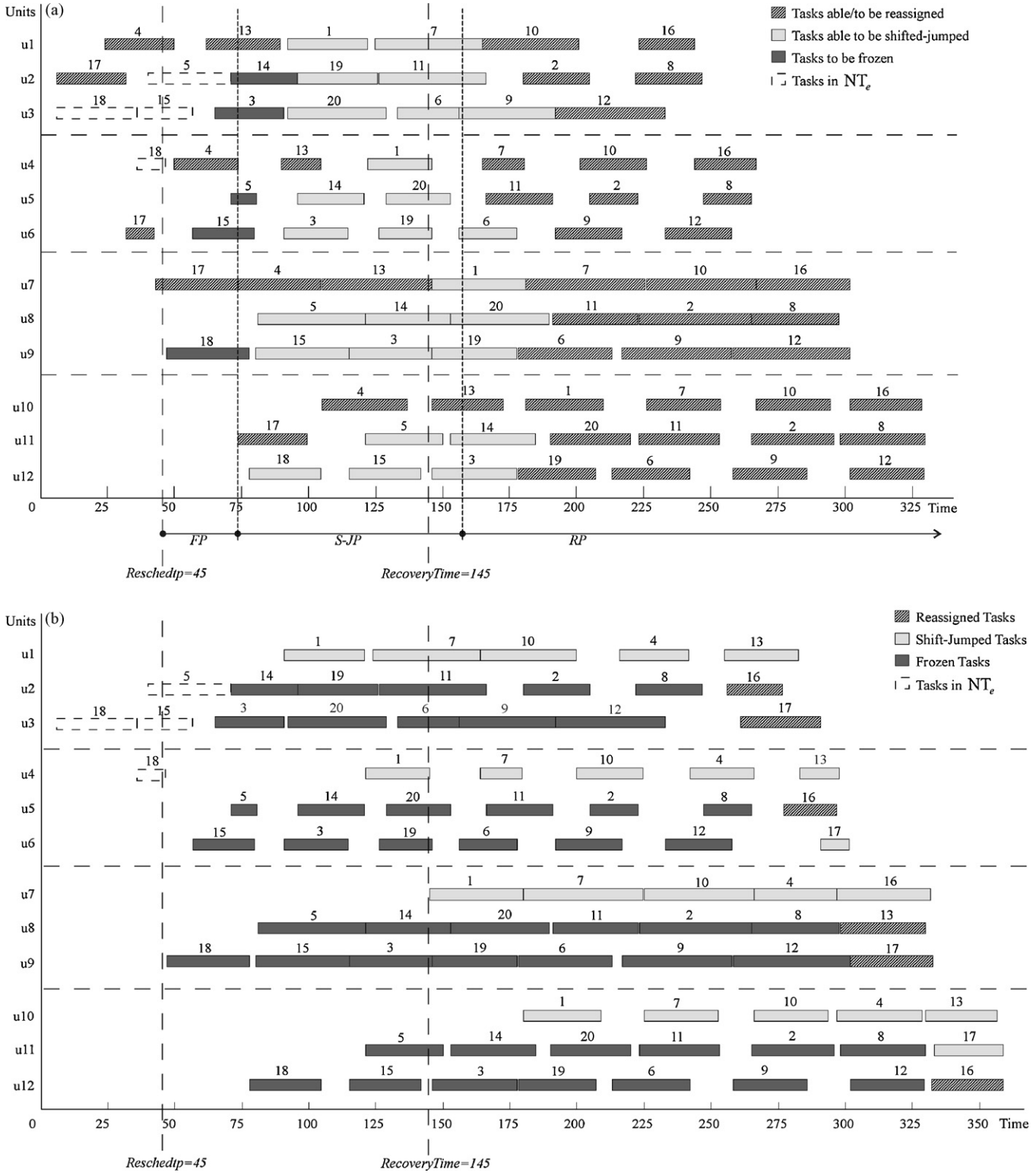

Fig. 10. Case Study 3—Scenario 5. (a) Problem specification showing the task classification and the associated action types. (b) Gantt diagram of the solution obtained when minimizing Makespan.

The same behavior was observed when different interstage storage policies were adopted, as shown in Table 6, which compares solutions obtained minimizing Makespan and *TD* when the plant operates under the NIS-UW policy.

Table 4 reveals that, in almost all cases, the worst values of Makespan were associated with the solutions corresponding to Scenario 1, under which the non-affected tasks are frozen. Naturally, these solutions try to preserve the structure of the original schedule and exhibit good values of *NST* and *NES*. On the other hand, when more flexibility is given to the rescheduling problem, lower *Mk* values are obtained. In fact, Scenarios 3 and 4, which allow more tasks to be reassigned, exhibit better values of Makespan, which are attained at the expense of low values of *NST* and *NES*. Moreover, it is worth to remark that solutions corresponding to scenarios 5 and 6 exhibit a good compromise between Makespan and the stability indexes. Fig. 10a presents a Gantt diagram that graphically shows the classification of tasks and action types to be applied to the example corresponding to Case Study 3—Scenario 5. In turn, Fig. 10b portrays the Gantt diagram obtained after solving the resulting rescheduling problem, with the minimization of Makespan as the objective function.

A comparative analysis of Fig. 10a and b shows that out of the 46 tasks that were allowed to be reassigned, only 6 were actually relocated in other equipment items. The other 40 were either shifted or frozen. In other words, most of the tasks that could be reassigned remained in their original units and some even did not move. Similarly, many tasks that were associated with a *Shift-JumpAT*, were not modified at all. In fact, the solution depicted in Fig. 10b shows that very few changes took place in the schedule. In consequence, the stability indexes associated with this solution (*TD* = 1732, *NST* = 0.53, *NES* = 0.79) are reasonably good. It should be remarked that regardless of these satisfactory stability properties, the solution exhibits a reasonable Makespan value of 361 time units, which is just 8 units higher that the optimal and suboptimal values obtained for Scenarios 3 and 4, in which tasks are given more freedom to be reassigned.

The solution of a wide variety of examples has shown us that solution strategies similar to the ones of scenarios 5 and 6 represent a natural way of taking into account stability aspects along with the optimization of regular performance measures, which otherwise could be quite difficult to include in a multiobjective function. In fact, the various metrics usually render values that may differ in several orders of magnitude. Besides, they are inherently difficult to normalize and, thus, hard to combine. For these reasons, the idea of considering different time periods along the scheduling horizon, and selecting the task-action types according to their memberships to those periods, is a natural way for dealing with stability while optimizing any regular performance measure. Since solutions are obtained in low CPU times, several scenarios corresponding to different values of *FP* and *S-JP*, can be solved for a particular problem instance, giving the scheduler various alternatives to choose his/her agenda.

Alternative settings can also be generated for Scenarios 1, 2, and 3, which have associated *Shift-JumpAT* action types, by varying the values of the $\alpha_{(i,s)}$ and $\beta_{(i,s)}$ parameters, and thus the values of $ls_{(i,s)}$ and $rs_{(i,s)}$. Problem specifications in which $ls_{(i,s)}$ and $rs_{(i,s)}$ are given values considerably lower than $lsmax_{(i,s)}$ and $rsmax_{(i,s)}$, will bound the task movements and generate more stable solutions. However, if the values are too small, such a specification may lead to an unfeasible condition having no solution. This situation occurred for Scenario 1 of Cases 1 and 3 (see Table 4), in which the value of $\beta_{(i,s)}$ had to be increased to reach feasible solutions. To assess the effect of these parameter values on the characteristics of the obtained solution, the results presented in Table 7 can be analyzed. This table shows solutions obtained for Case study 1—Scenario 2 (Tasks in RT_e^{IA} and RT_e^{NA} are associated with *Shift-JumpAT* action types), when minimizing Makespan. In these problem instances, for all the tasks in RT_e^{IA} and RT_e^{NA}, $ls_{(i,s)}$ was set equal to $lsmax_{(i,s)}$. In turn, $rs_{(i,s)}$ was specified according to expression (24), with different values given to $\beta_{(i,s)}$. As seen in the first row, a very low value of $\beta_{(i,s)}$ led to an unfeasible problem. However, as more flexibility was given to the possible task movements, solutions having better Makespan values were reached and the trade-off between *Mk* and *TD* was made explicit.

Table 7
Case study 1—Scenario 2, NIS-ZW policy, Makespan minimization. Effect of $\beta_{(i,s)}$ on the trade-off between *Mk* and *TD*.

$\beta_{i,s}$	Mk	TD	TCT	NST	NES
1			Unfeasible solution		
2	367	1875	4859	0.24	0.96
3	363	2591	4846	0.24	0.96
4	356	4216	4722	0.13	0.96
5	356	4264	4734	0.13	0.96
6	356	4280	4738	0.13	0.96
7	356	3735	4748	0.24	0.96
8	355	4196	4721	0.18	0.96
9	355	4196	4721	0.18	0.96

By assigning distinct values to the $\alpha_{(i,s)}$ and $\beta_{(i,s)}$ parameters, solutions exhibiting different trade-offs between regular performance measures and stability ones can be obtained. However, it is not straight forward to fix these values to obtain solutions properly balancing these different objectives. In cases where the scheduler does not have enough knowledge to fix these parameter values, and provided solutions are reached with almost no computational effort, various alternative repaired schedules can be obtained in a very short time by iteratively modifying the values of $\alpha_{(i,s)}$ and $\beta_{(i,s)}$. Then, the scheduler can examine the solutions and choose his/her preferred one. A similar approach can be followed if a solution approach based on the definition of several time periods is adopted. In case the scheduler does not have enough know-how to fix the length of the *FP* and *S-JP* periods, several alternative solutions can be quickly generated by iteratively modifying the length of these periods.

7. Conclusions and future work

This contribution introduced a support framework, aimed at addressing the repair-based reactive scheduling problem of batch plants. This framework has been envisioned to operate under an event-driven rescheduling policy. It is based on a hybrid representation, which combines an explicit object-oriented domain model and a constraint programming (CP) approach.

The framework is able to capture the status of the in-progress schedule, and to typify the unexpected event in order to characterize its context and impact. This allows making a proper specification of the rescheduling problem to be faced. Tasks to be rearranged are recognized and automatically classified into different categories. Then, the set of the most suitable rescheduling action types (e.g. shift-jump, reassign, freeze, etc.) is specified for them. Since a given specification many lead to several solutions, the second stage of the approach relies on a CP model to address the problem just defined. To create such model, action types are automatically transformed into constraints. Provided that good quality solutions can be reached in very low CPU times, alternative solution scenarios focusing on stability and regular performance measures can be posed by the scheduler for each problem. In this way, the scheduler can analyze the possible rescheduling solutions developed with the framework and select the most suitable one for the problem being faced.

The proposed solution methodology attempts to get together the benefits of a repair-based method (limited schedule disrup-

tion and low computational requirements), with the advantages a complete rescheduling approach (non-myopic, overall view of the rescheduling system). The current implementation of the proposal is oriented towards multiproduct stage batch plants, operating under a batch-based approach, and with UIS, NIS/UW, NIS/ZW interstage storage and operational policies. It is limited to a set of unforeseen events (unit failure and performance modification, batch cancellation/modification/new arrival, inclusion of additional tasks). However, the underlying rationale of the system would allow extending it in the future to consider multipurpose batch plants, other interstage and operational policies and a wider range of disruptive events (e.g. modification of non-renewable resources availability).

Another future work is concerned with determining the need for rescheduling. Disrupting events can cause significant disturbances, mild disruptions, or just have a negligible impact, requiring minor adjustments of task timings. In its current version, the framework assumes that the event triggering the rescheduling process causes a major disturbance, and can only assist in the evaluation of the event impact by checking if the schedule became unfeasible or not. However, a much clever assistance is necessary. Unfortunately, this issue has not been addressed in the literature yet and we plan to tackle it in the future. Having an explicit representation of the production environment, the in-progress schedule and the event, we have the foundations to develop intelligent capabilities, able to assess the event impact and determine the associated response accordingly.

Acknowledgments

The authors wish to acknowledge the financial support received from CONICET (PIP 2754), UNL (CAI + D 2009), and ANPCyT (PAE-PICT-2007-00051).

References

ANSI/ISA-88 (1995). *Batch control part 1: Models and terminology*.
ANSI/ISA-88 (2001). *Batch control part 2: Data structures and guidelines for languages*.
ANSI/ISA-88 (2001). *Batch control part 3: General and site recipe models and representations*.
Aytug, H., Lawley, M. A., McKay, K., Mohan, S., & Uzsoy, R. (2005). Executing production schedules in the face of uncertainties: A review and some future directions. *European Journal of Operational Research*, *161*, 86–119.
Booch, G., Rumbaugh, J., & Jacobson, I. (1999). *UML: The unified modelling language user guide*. Addison-Wesley.
Castro, P. M., & Grossmann, I. E. (2005). New continuous-time MILP model for the short-term scheduling of multistage batch plants. *Industrial and Engineering Chemistry Research*, *44*, 9175–9190.
Cott, B. J., & Macchietto, S. (1989). Minimizing the effects of batch process variability using online schedule modification. *Computers and Chemical Engineering*, *13*, 105–113.
Hasebe, S., Hashimoto, I., & Ishikawa, A. (1991). A general reordering algorithm for scheduling of batch processes. *Journal of Chemical Engineering of Japan*, *24*, 483–489.
Henning, G. H. (2009). Production scheduling in the Process Industries: Current trends, emerging challenges and opportunities. In R. de Brito Alves, C. Oller do Nascimento, & E. Biscaia (Eds.), *Computer-Aided Chemical Engineering-27*. Elsevier Science Ltd.
Huercio, A., Espuña, A., & Puigjaner, L. (1995). Incorporating on-line scheduling strategies in integrated batch production control. *Computers and Chemical Engineering*, *19*, S609–S615.
Ierapetritou, M. G., & Floudas, C. A. (1998). Effective continuous-time formulation for short-term scheduling: I. Multipurpose batch processes. *Industrial and Engineering Chemistry Research*, *37*, 4341–4359.
ILOG (2000). *ILOG Scheduler 5.0 User's Manual*. France: ILOG.
ILOG (2002). *ILOG OPL Studio 3.5 User's Manual*. France: ILOG.
Janak, S. A., Floudas, C. A., Kallrath, J., & Vormbrock, N. (2006). Production scheduling of a large-scale industrial batch plant. II. Reactive scheduling. *Industrial and Engineering Chemistry Research*, *45*, 8253–8269.
Kanakamedala, K. B., Reklaitis, G. V., & Venkatasubramanian, V. (1994). Reactive scheduling modification in multipurpose batch chemical plants. *Industrial and Engineering Chemistry Research*, *33*, 77–90.
Kelleher, G., & Cavichiollo, P. (2001). Supporting rescheduling using CSP, RMS and POB—An example application. *Journal of Intelligent Manufacturing*, *12*, 343–357.
Kopanos, G. M., Capón-García, E., Espuña, A., & Puigjaner, L. (2008). Costs for rescheduling actions: A critical issue for reducing the gap between scheduling theory and practice. *Industrial and Engineering Chemistry Research*, *47*, 8785–8795.
Li, Z., & Ierapetritou, M. (2008). Process scheduling under uncertainty: Review and challenges. *Computers and Chemical Engineering*, *32*, 715–727.
Méndez, C. A., & Cerdá, J. (2003). Dynamic scheduling in multiproduct batch plants. *Computers and Chemical Engineering*, *27*, 1247–1259.
Méndez, C. A., & Cerdá, J. (2004). An MILP framework for batch reactive scheduling. *Computers and Chemical Engineering*, *28*, 1059–1068.
Musier, R. F. H., & Evans, L. (1990). Batch process management. *Chemical Engineering Progress*, *87*, 66–77.
OMG (2006). *Object Constraint Language*. OMG Available Specification. Version 2.0. formal/06-5-01. Available in: http://www.omg.org/spec/OCL/2.0/PDF.
Ouelhadj, D., & Petrovic, S. (2009). A survey of dynamic scheduling in manufacturing systems. *Journal of Scheduling*, *12*, 417–431.
Roslöf, J., Harjunkoski, I., Björkqvist, J., Karlsson, S., & Westerlund, T. (2001). An MILP-based reordering algorithm for complex industrial scheduling and rescheduling. *Computers and Chemical Engineering*, *25*, 821–828.
Sanmartí, E., Huercio, A., Espuña, A., & Puigjaner, L. (1996). A combined scheduling/reactive scheduling strategy to minimize the effect of process operations uncertainty in batch plants. *Computers and Chemical Engineering*, *20*, 1263–1268.
Smith, S. F. (1995). In D. E. Brown, & W. T. Scherer (Eds.), *Reactive scheduling systems. Intelligent Scheduling Systems*. Kluwer Press.
Vieira, G., Herrmann, J., & Lin, E. (2003). Rescheduling manufacturing systems: A framework of strategies, policies, and methods. *Journal of Scheduling*, *6*, 39–62.
Vin, J. P., & Ierapetritou, M. (2000). A new approach for efficient rescheduling of multiproduct batch plants. *Industrial and Engineering Chemistry Research*, *39*, 4228–4238.
Zeballos, L. J., Novas, J. M., & Henning, G. P. A CP Formulation for Scheduling Multiproduct Multistage Batch Plants, *Computers and Chemical Engineering*. Under revision.