Contents lists available at SciVerse ScienceDirect

Computers and Chemical Engineering

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

# *SmartGantt* – An interactive system for generating and updating rescheduling knowledge using relational abstractions

Jorge Palombarini[a], Ernesto Martínez[b,*]

[a] GISIQ (UTN), Av. Universidad 450, Villa María 5900, Argentina
[b] INGAR (CONICET-UTN), Avellaneda 3657, Santa Fe S3002 GJC, Argentina

ARTICLE INFO

*Article history:*
Received 2 February 2012
Received in revised form 15 June 2012
Accepted 18 June 2012
Available online 30 June 2012

*Keywords:*
Batch plant management
Cognitive production systems
Manufacturing control
Rescheduling
Relational reinforcement learning
Uncertainty

ABSTRACT

Generating and updating rescheduling knowledge that can be used in real time has become a key issue in reactive scheduling due to the dynamic and uncertain nature of industrial environments and the emergent trend towards cognitive systems in production planning and execution control. Disruptive events have a significant impact on the feasibility of plans and schedules. In this work, the automatic generation and update through learning of rescheduling knowledge using simulated transitions of abstract schedule states is proposed. An industrial example where a current schedule must be repaired in response to unplanned events such as the arrival of a rush order, raw material delay, or an equipment failure which gives rise to the need for rescheduling is discussed. A software prototype (*SmartGantt*) for interactive schedule repair in real-time is presented. Results demonstrate that responsiveness is dramatically improved by using relational reinforcement learning and relational abstractions to develop a repair policy.

## 1. Introduction

Increasing global competition, a shift from seller markets to buyer markets, mass customization, operational objectives that highlight customer satisfaction and the need to ensure a high level of efficiency in production systems give rise to a complex shop-floor dynamics due to unplanned and disruptive events in industrial environments (Henning & Cerdá, 2000; Zaeh, Reinhart, Ostgathe, Geiger, & Lau, 2010). Moreover, stringent requirements with regard to reactivity, adaptability and traceability in production systems, and by extension in supply chains, are demanded for products and processes by both suppliers and clients all over the product lifecycle.

In order to deal with the above challenges, is necessary to achieve higher degrees of flexibility, adaptability, autonomy and learning capabilities in production systems. Disruption management, self-reconfiguration and adaptive behavior are key capabilities in order to fulfill the aforementioned requirements without sacrificing cost effectiveness, product quality and on-time delivery. In this context, established production planning and control systems are vulnerable to unplanned events and intrinsic variability of a manufacturing environment where difficult-to-predict circumstances occur as soon as schedules are released to the shop-floor. Equipment failures, quality tests demanding reprocessing operations, rush orders, delays in material inputs from previous operations and arrival of new orders give rise to uncertainty in real time schedule execution. Hence, schedules generated under the deterministic assumption are often suboptimal or even infeasible (Li & Ierapetritou, 2008; Vieira, Herrmann, & Lin, 2003; Zaeh et al., 2010). As a result, reactive scheduling is heavily dependent on the capability for generating and representing knowledge about strategies for repair-based scheduling in real-time. Moreover, timely producing satisfactory schedules rather than optimal ones, in reasonable computation time and fully integrated with enterprise resource planning and control systems is mandatory for responsiveness and agility (Trentesaux, 2009).

Existing literature related to reactive scheduling mainly aims to exploit peculiarities of a specific problem structure (Adhitya, Srinivasan, & Karimi, 2007; Miyashita & Sycara, 1994; Miyashita, 2000; Zhu, Bard, & Yu, 2005; Zweben, Davis, Doun, & Deale, 1993). More recently, Li and Ierapetritou (2008) have incorporated uncertainty in the form of a multi-parametric programming approach to generate rescheduling knowledge for specific events. However, the tricky issue is that resorting to a feature-based representation of schedule states is very inefficient, and generalization to unseen schedule states is highly unreliable. Therefore, transferring heuristics or a rescheduling policy is difficult to unseen scheduling domains, being the user-system interactivity severely affected due to the need of compiling the repair-based strategy for each disruptive event separately.

* Corresponding author. Tel.: +54 342 4534451; fax: +54 342 4553439.
*E-mail address:* ecmarti@santafe-conicet.gob.ar (E. Martínez).

0098-1354/$ – see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.compchemeng.2012.06.021

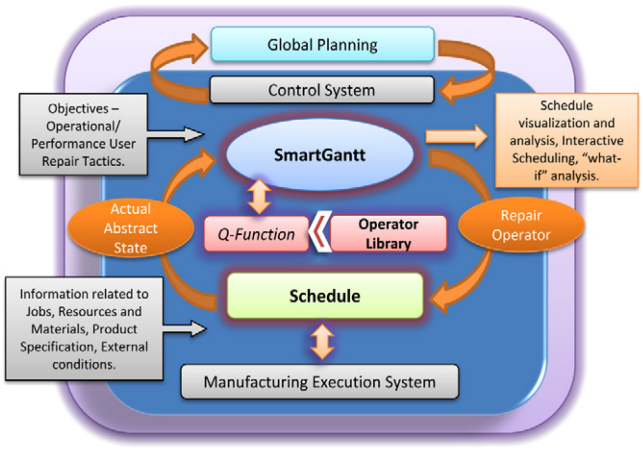

Global Planning
Control System
Objectives – Operational/ Performance User Repair Tactics.
SmartGantt
Schedule visualization and analysis, Interactive Scheduling, "what-if" analysis.
Actual Abstract State
Q-Function
Operator Library
Repair Operator
Information related to Jobs, Resources and Materials, Product Specification, External conditions.
Schedule
Manufacturing Execution System

**Fig. 1.** Repair-based architecture implemented by *SmartGantt*.

Most of the existing works on rescheduling prioritize rescheduling *efficiency* using a mathematical programming approach (Li & Ierapetritou, 2008). However, schedule *stability* is also an important objective that must be accounted for when generating a rescheduling policy (Pfeiffer, Kádár, & Monostori, 2007; Rangsaritratsamee, Ferrell, & Kurz, 2004). Moreover, the type of abstractions used for representing schedule states and repair actions is of paramount importance to scale up solutions found for small-size case studies to industrial applications involving thousands of tasks and hundreds of resources in an uncertain environment. In particular, humans can succeed in rescheduling thousands of tasks and resources by increasingly learning a repair strategy using a natural abstraction of a schedule situation or state: a number of objects (tasks and resources) with attributes and relations (precedence, synchronization, etc.) among them. First-order relational representations enable exploiting the existence of domain objects, of relations (or, properties) over these objects, and make room for quantification over objectives (goals), action effects and abstract properties of schedule states (Blockeel, De Raedt, Jacobs, & Demoen,1999). The very success of Gantt charts (Wilson, 2003) as a support tool for (re)scheduling tasks at the shop-floor level is that they provide a ready visualization of precedence and synchronization relationships between tasks and resources over a common time line.

In this work, a novel real-time rescheduling prototype application called *SmartGantt* based on a relational (deictic) representation of (abstract) schedule states and repair operator is presented. To learn a near-optimal policy for rescheduling using simulations of schedule state transitions (Croonenborghs, 2009), an interactive repair-based strategy bearing in mind different goals and disruptive events is proposed. To this aim, domain-specific knowledge for reactive scheduling is generated and updated using relational reinforcement learning (RRL) (Džeroski, De Raedt, & Driessens, 2001) and relational abstractions (De Raedt, 2008).

## 2. Repair-based (re)scheduling in *SmartGantt*

Fig. 1 depicts the repair-based architecture implemented by *SmartGantt*, embedded in a more general setting that includes an enterprise resource planning (ERP) system and a manufacturing execution system (MES) along with a communication and control infrastructure. *SmartGantt* also integrates artificial cognitive capabilities in resources and processes through a human–agent–machine interface to favor achieving by design the type of flexibility and adaptability that are needed in production systems (Trentesaux, 2009). The mentioned infrastructure consists of the system control level, the process control level and the planning level. The system control level is responsible for the physical execution of the production process. The global planning level administrates, coordinates and dispatches the incoming orders to the production system, and also involves transport control and planning/scheduling features. Transport control is responsible for the material supply to the production system whereas planning/scheduling capabilities are integrated in *SmartGantt*. The respective order data (e.g. slack time), the current boundary conditions (e.g. machine availability) and the overall system utilization (e.g. capacity utilization) are the prerequisites to decide when an order is released to the shop-floor. Based on the specification of the order and information related to operational objectives, tasks, resources and materials, *SmartGantt* allocates the respective production operations to feasible resources in the production system and the needed resources are booked based on estimated production times. Later on, specific knowledge about resource states (e.g. capability profiles) as well as the respective product-related (e.g. quality) and process-related data (e.g. production steps) are used for feasible schedule modifications (e.g. due to a rework operation) and the on-going repair sequence optimization is applied to face unforeseen events (e.g. machine breakdowns) in an autonomic way.

In *SmartGantt*, rescheduling knowledge about optimal selection of repair operators towards a goal state is generated through reinforcements using a simulator of schedule state transitions. In the simulation environment, an instance of the schedule is interactively modified by the learning system which executes control actions using a sequence of repair operators until a repair goal is achieved. In each learning episode, *SmartGantt* receives information from the current schedule situation or state $s$, and then selects a repair operator $a$, which is applied to the current schedule, resulting in a new one. The evaluation of the resulting quality of a schedule after a repair operator has been used by *SmartGantt* is performed using the simulation environment via an objective or reward function $r(s)$. The learning system then updates its action-value function $Q(s,a)$ that estimates the value or utility of resorting to the chosen repair operator $a$ in a given schedule state $s$. Value updates are made using a reinforcement learning algorithm (Sutton & Barto, 1998), such as the following $Q$-learning rule

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_b Q(s',b) - Q(s,a)] \quad (1)$$

where $s'$ is the resulting state of using the repair operator $a$ at the schedule state $s$. Accordingly, a repair operator in a given schedule state is chosen based on the *optimal policy*: $a_i = \pi^*(s_i)$ if $a_i$ has the highest $Q$-value in the state $s_i$. A simulation-based algorithm to learn the optimal policy is the well-known $Q$-learning (Sutton & Barto, 1998; Watkins, 1989).

Based on learning episodes, the $Q$-learning algorithm updates these $Q$-values incrementally while the reinforcement learning agent interacts with a real or simulated environment. In this work, the relational variant of $Q$-learning is used (see next section for details). The main benefit of applying reinforcement learning techniques, such as $Q$-learning algorithm, in automated generation of rescheduling knowledge for improving stability and efficiency of real-time reactive scheduling is that there is no extra burden on domain experts, allows online adaptation to a dynamic environment and make room for relational abstractions that can be used to deal with large state spaces, e.g. in supply chains or distributed plant scheduling. By accumulating enough experiences over many simulated transitions between schedule states, *SmartGantt* is able to learn an optimal policy for choosing the best repair operator at each schedule state in the path to a goal state (repaired schedule).

For repairing a schedule, *SmartGantt*, is given a repair-based *goal* function *goal:S* $\rightarrow$ {*true*, *false*} defining which states in the schedule are target states in a repaired schedule, e.g. states where Total

```
Initialize the Q-function hypothesis $\hat{Q}_0$
$e \leftarrow 0$
repeat
    Examples $\leftarrow \varnothing$
    Generate a starting schedule state $s_0$
    $i \leftarrow 0$
    repeat
        choose a repair operator $a_i$ at $s_i$ using a policy (e.g., ε-greedy) based on the current
        hypothesis $\hat{Q}_e$ implement operator $a_i$, observe $r_i$ and the resulting schedule $s_{i+1}$
        $i \leftarrow i+1$
    until schedule state $s_i$ is a goal state
    for $j = i-1$ to 0 do
        generate example $x = (s_j, a_j, \hat{q}_j)$, where $\hat{q}_j \leftarrow r_j + \gamma \max_a \hat{Q}_e(s_{j+1}, a)$
        Examples $\leftarrow$ Examples $\cup \{x\}$
    end for
    Update $\hat{Q}_e$ to $\hat{Q}_{e+1}$ using Examples and a relational regression algorithm (e.g. TG)
until no more learning episodes
```

**Fig. 2.** A RRL algorithm for learning to repair schedules using simulations.

Tardiness is less than or equal to 1 working day. The objective of any schedule repair task can be phrased as:

"Given a starting state for the schedule $s_1$, find a sequence of repair operators $a_1, a_2, \ldots, a_n$ *with* $a_i \in A$ such that *goal*$(\delta(\ldots\delta(s_1, a_1)\ldots, a_n))$ = *true*, where $\delta$ is the transition function which is unknown to the learning agent."

Also, a reward function is used as a guideline to learn a repair policy from reinforcements based on simulated state transitions (Martínez, 1999). The reward function is unknown to the learning agent and depends on the repair goal selected. Resorting to the reward function and simulations, the repair *policy*: $a_i = \pi^*(s_i)$ can be approximated using a regression tree to advantage (Blockeel & De Raedt, 1998; Van Otterlo, 2009), and then used to compute on the fly the repair sequence needed to reach a schedule that satisfy the goal stated by an external user. Reactive repair of schedules improves responsiveness at the shop-floor to handle unplanned events and disturbances that make the current schedule unfeasible (Palombarini & Martínez, 2010). Also, different operational and performance objectives, as well as user preferences and customized repair tactics can be provided interactively by a human expert to *SmartGantt*. The user agent interface provides visualization, alternative solutions and "*what-if*" analysis capabilities, so that the user can work in a fully interactive fashion that highlights different views of a rescheduling task and parameters that may influence the solution found.

## 3. Relational reinforcement learning

Relational reinforcement learning (RRL) is often formulated in the formalism of Relational Markov Decision Processes (RMDP), which are an extension from standard MDPs based on relational representations in which states correspond to Herbrand interpretations (Džeroski et al., 2001). Relational modeling offers many possibilities for abstraction due to the structured form of ground atoms in states and actions. Thus, RRL algorithms are concerned with reinforcement learning in domains that exhibit structural properties and in which different kinds of related objects such as tasks and resources exist (De Raedt, 2008; Van Otterlo, 2009). This is usually characterized by a large and possibly unbounded number of different states and actions as it is the case of planning and scheduling worlds. RRL stores the *Q*-values in a logical regression tree (Blockeel & De Raedt, 1998) and its relational version of the *Q*-learning algorithm is shown in Fig. 2. So, in RRL states are represented as sets of first-order logical facts, and the learning algorithm can only see one state at a time. Actions are also represented relationally as predicates describing the action as a relationship between one or more variables, as it is shown in Example 1.

**Example 1.** **state1** = {totTard(53.86), maxTard(21.11), avgTard(3.85), totaWIP(46.13), resLoad(0,30.39), resLoad(1,47.93), resLoad(2,21.68), tRatio(3.34), invRatio(6.06), resTard(0,6.12), resTard(1,39.57), resTard(2,8.16), totalCT(3), focalRSwap, focalLSwap, focalAltRSwap, focalAltLSwap, matArriv(0), lTard(0), rTard(6.075), focalTask(task(task14)), resource(0,extruder,[task(task13,0,1.1,1.1,a,0,11,110),task(task14,1.1,11.05,9.95,a,0.05,11,995)...]), resource(1,extruder,[task(task11,0,5.66,5.66,c,0,15,849),task(task6,5.66,7.26,1.6,c,0,10,240)...]), resource(2,extruder,[task(task12,0,2.31,2.31,b,0,18,346))...};
**action1** = action(leftJump(task(task14),task(task13))).

In Example 1 a set of relational facts are shown to represent a schedule state named **state1** and a relational fact **action1** which correspond to the repair operator applied in such abstract state. The relational state is made up of logical functors which express relations between objects or attributes of them. Some relations involve real values related to the global schedule state, such as `totTard`, `maxTard` and `avgTard` which represent the Total, Maximum and Average Tardiness associated with the current schedule. Additionally, more complex functors have been designed to reflect the workload and tardiness per resource (`resLoad` and `resTard` functors) which have two arguments: resource ID and a real value representing the Workload or Total Tardiness in a given resource, respectively. The deictic feature of the relational representation is specified using the `focal` functor, which has only one argument of the type `task`, stating which task is taken as the focal point for repair. The focal task fixes the values of the `ltard` and `rtard` functors which specify the amount of tardiness accumulated by the tasks which are programmed to start before and after the focal task, respectively. Also, the selection of the focal task generates a sub-set of logical atoms which enables the system to evaluate which possible repair operations can be carried out in a particular state. An example of such atoms is `focalRSwap` which indicates that is possible to make a swap operation between the focal task and another task which is programmed to start after the first, and in the same resource. Also, the `resource` functor has tree arguments: resource ID, resource name, and a list of tasks which have been assigned to it. The order in the list also represents the execution order of the tasks. Each element in the list is a `task` functor, with the arguments Task Name, Start Time, Finish Time, Total Processing Time, Product Type, Tardiness, Due Date and Total Lbs. Finally, the repair operation applied in the state is represented using the `action` functor. It takes as an argument the repair operator, which in the example is `leftSwap`, parameterized with the focal task as the first argument and another task as the second argument to describe the swapping operation completely.

The computational implementation of the RRL algorithm has to deal successfully with the relational format for (*states*, *actions*)-pairs in which rescheduling training examples are represented. The learning agent is given a continuous stream of (*state*, *action*, *q-value*)-triplets and has to predict *q-values* for (*state*, *action*)-pairs during simulation-based learning. As a result, the *Q*-value function is obtained and the repair policy is defined.

For completeness and efficiency, the relational representation of states and actions along with the inductive logic programming component of the RRL algorithm requires resorting to a body of *background knowledge* which is generally true for the entire domain and thus facilitates induction. After the *Q*-function hypothesis has been initialized, the RRL algorithm starts running learning episodes (Džeroski et al., 2001; Sutton & Barto, 1998). The algorithm presents the set of (*state*, *action*, *q-value*)-triplets encountered in each learning episode to a relational regression engine, which in turn will

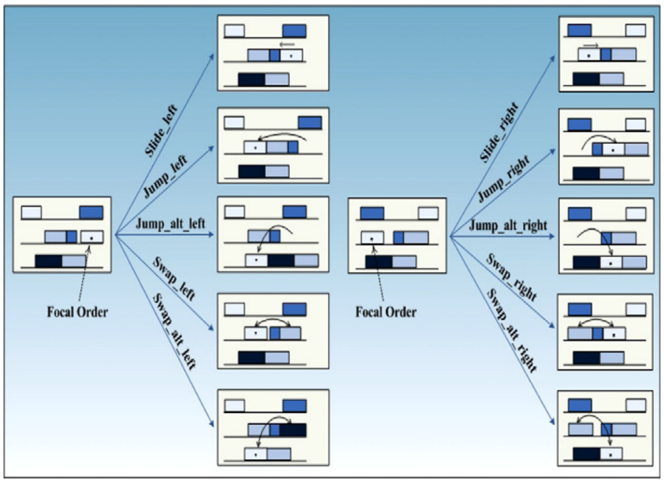


**Fig. 3.** Deictic repair operators.

use this set of examples to update the current regression tree that encode the *Q*-function. The TG relational regression algorithm (De Raedt, 2008; Džeroski et al., 2001) is an evolution of the G algorithm proposed in Chapman and Kaelbling (1991) and is used by *SmartGantt* for encoding simulated experience in a compact, yet reactive decision-making rule for generating a sequence of repair operators available at each schedule state *s* in response to events and disturbances.

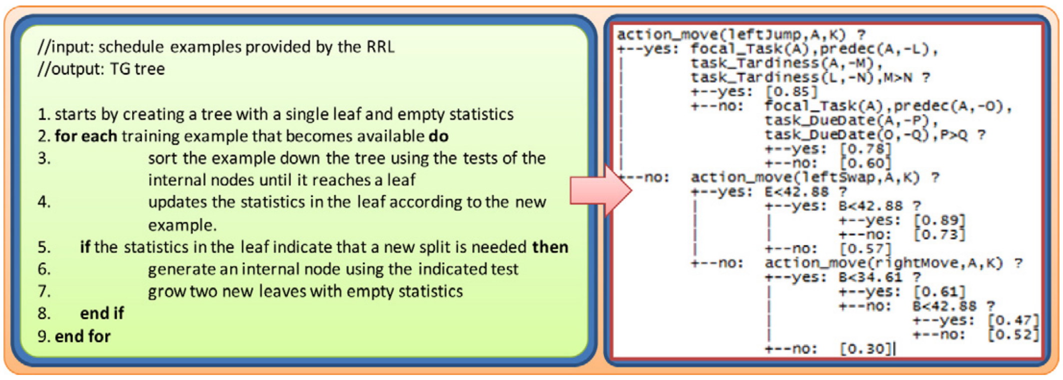
Drawbacks of attribute-value (propositional) representations of scheduling states that have been described in previous sections, are solved in *SmartGantt* by means of *relational* (or *first-order*) deictic representations such that each state can be characterized by only those facts that hold in it. In a deictic representation, both scheduling states and repair operators (actions) are defined in relation to a given focal point (i.e. a task, an order or a resource) as shown in Fig. 3. These local repair operators move the position of a task alone, however due to the ripple effects caused by tight resource-sharing constraints other tasks must be moved as well, which is often not desirable. Whenever the goal-state for the schedule cannot be achieved using primitive repair operators, more elaborated macro-operators can be used to implement a combination of basic repair operators such as `task-swapping`, `batch-split` or `batch-merge` until a goal state in the repaired schedule (e.g. order insertion without delaying other orders) is achieved.

To exploit the inherent structure of schedule states, as well as the structure shared among several "similar" schedules which is instrumental in reducing the state space and accelerate learning, *SmartGantt* performs an induction process over seen sequences of schedule states and repair operators, obtaining a set of **Abstract States** (AS), which are conjunctions$\equiv 1\Lambda \ldots \Lambda\ m$ of logical atoms (e.g. a logical query). Thus, an AS is basically a logical sentence, specifying general properties of several states visited during learning to reschedule through simulated transitions. Thus, *Q*-function relies on a set of abstract states that together encode the kind of rescheduling knowledge learned through trial-and-error learning using repair operators and evaluative feedback. This knowledge can be used in real time to repair plans that have been affected by disruptive events. Furthermore, an AS definition is independent of the kind of disruptive event which may have caused it: it only depends on the desired goal for the repaired schedule state. As a result, is of marginal importance to identify the event type that has driven the schedule to the current state, but to find a sequence of repair operators to achieve a goal for a repaired schedule. This fact readily allows transferring the repair policy to different problems were similar relations are present without any further learning. An abstract state *S*, covers a ground state *s* iff $s|=S$, which is decided by *SmartGantt* using *θ-subsumption* (Driessens & Ramon, 2003).

## 4. Knowledge representation

As it was previously mentioned, the repair policy learned using RRL and the abstract schedule states are stored in a first order logical decision tree. In logical decision trees, an example is a Prolog knowledge base described by a set of facts (as described in Example 1 above) where each example corresponds to a state description in the schedule domain, and tests in the internal nodes of the trees are (Prolog)-queries e.g. `precedes`(**A**,**B**)? (is there any task **A** which precedes a task **B** in the resource?). Since the outcome of a query is either true or false, the resulting trees are always binary. Furthermore, the queries can contain variables, and these variables may be shared among several nodes in a tree. When variables are shared among several nodes they refer to the same object. The semantic of the tree is completely characterized by the corresponding Prolog program. To classify an example one tries to assess whether the condition part of the first rule is satisfied. If it is, the corresponding prediction is used; otherwise the second rule is tried. In this work the TG algorithm (Blockeel et al., 1999; Driessens, Ramon, & Blockeel, 2001) depicted in Fig. 4 is used to generate the tree which represents the repair policy. TG is a combination of the Top Down Induction of Logical Decision Trees (TILDE) algorithm (Blockeel & De Raedt, 1998), that constructs a first order representation for classification and regression trees using an inductive logic programming approach, and the algorithm G (Chapman & Kaelbling, 1991) that uses a number of statistical values related to the performance of each possible extension for each leaf of the tree to build it incrementally.

Similar to TILDE, TG uses a relational representation language to describe the examples and tests that can be used in a regression

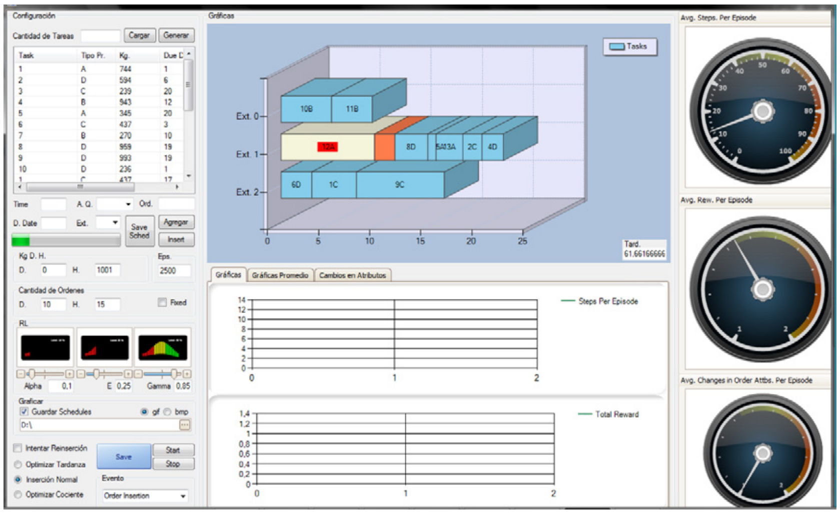


```
//input: schedule examples provided by the RRL
//output: TG tree

1. starts by creating a tree with a single leaf and empty statistics
2. for each training example that becomes available do
3.        sort the example down the tree using the tests of the
          internal nodes until it reaches a leaf
4.        updates the statistics in the leaf according to the new
          example.
5.    if the statistics in the leaf indicate that a new split is needed then
6.        generate an internal node using the indicated test
7.        grow two new leaves with empty statistics
8.    end if
9. end for
```

```
action_move(leftJump,A,K) ?
+--yes: focal_Task(A),predec(A,-L),
|       task_Tardiness(A,-M),
|       task_Tardiness(L,-N),M>N ?
|       +--yes: [0.85]
|       +--no:  focal_Task(A),predec(A,-O),
|               task_DueDate(A,-P),
|               task_DueDate(O,-Q),P>Q ?
|               +--yes: [0.78]
|               +--no:  [0.60]
+--no:  action_move(leftSwap,A,K) ?
        +--yes: E<42.88 ?
        |       +--yes: B<42.88 ?
        |       |       +--yes: [0.89]
        |       |       +--no:  [0.73]
        |       +--no:  [0.57]
        +--no:  action_move(rightMove,A,K) ?
                +--yes: B<34.61 ?
                |       +--yes: [0.61]
                |       +--no:  B<42.88 ?
                |               +--yes: [0.47]
                |               +--no:  [0.52]
                +--no:  [0.30]
```

**Fig. 4.** TG algorithm specification (left) and an extract of the resulting relational regression tree (right).



| Task | Tipo Pr. | Kg. | Due D |
|---|---|---|---|
| 1 | A | 744 | 1 |
| 2 | D | 594 | 6 |
| 3 | C | 239 | 20 |
| 4 | B | 943 | 12 |
| 5 | A | 345 | 20 |
| 6 | C | 437 | 3 |
| 7 | B | 270 | 10 |
| 8 | D | 959 | 19 |
| 9 | D | 993 | 19 |
| 10 | D | 236 | 1 |
| 1 | C | 437 | 17 |

Kg D. H.: D. 0 H. 1001
Eps. 2500
Cantidad de Ordenes: D. 10 H. 15 ☐ Fixed
RL: Alpha 0.1 E 0.25 Gamma 0.85
Graficar: ☑ Guardar Schedules [✓ gif] bmp
D:\
☐ Intentar Reinserción
○ Optimizar Tardanza
● Inserción Normal
○ Optimizar Cociente
Evento: Order Insertion

**Fig. 5.** SmartGantt graphical user interface.

tree. The RRL algorithm feeds TG with all the generated examples in one learning episode. TG tests the examples one by one through the tree until it reaches a leaf. The *Q*-value of the reached leaf is updated with the new value (the *Q*-value maintained at each leaf is the average of the *Q* values of all the examples that reach it). The statistics for each leaf consists of the number of examples on which each possible test succeeds or fails, as well as the sum of the *Q*-values and the sum of the squared *Q*-values for each of the two subsets created by a test. These statistics can be calculated incrementally and are sufficient to compute whether a test is significant or not, i.e. whether the variance of the *Q*-values of the examples would be reduced sufficiently by splitting the node using that particular test. If the test is relevant with a high confidence, then the leaf is split into two new leaves, and each of them receives a *Q*-value based on the statistics obtained from the test used to split its parent node. Later on, this value is updated as new examples are sorted in the leaf. The TG algorithm provides as output a tree with knowledge stored for repair actions *Q*-values at different abstract states.

Iteratively, the algorithm in Fig. 2 generates new examples and delivers them to the tree. Each sample is sorted through the tree until it reaches a leaf where it is stored and the leaf's *Q*-value is updated with the *Q*-value for the presented example (the *Q* value of the leaf is the average value of the *Q* values from the samples stored in the leaf). After a minimum sample size (MSS) of examples is stored in a leaf (such parameter can be established in the configuration of the algorithm), the TG algorithm applies all possible tests to this set of examples, and verifies which ones offer the best split of the examples into two subsets. The one offering the best split is chosen as the leaf's test and two new leaves with zero statistics are created from it. The resulting tree encodes the experience gained through successive simulated repairs, and is parsed by *SmartGantt* into a Prolog program which can be used in real time to repair schedules found in dealing with different events and disturbances.

A key issue in the generation of a regression tree using the inductive logic programming approach described previously is the use of background knowledge, which consists of the definitions of general predicates that can be used in the tests and subsequently in the generation of Prolog rules, which influences the concepts that are represented. Therefore, the background knowledge (*BK*) serves a number of purposes, but two of the most important are: (i) the extension of the formal language explained in previous sections with additional predicate definitions, e.g. as derived relations in planning (Shapiro, Langley, & Shachter, 2001) or background predicates and (ii) the provision of domain constraints and ramifications (i.e. the static and dynamic laws in the scheduling domain). For example, whereas a ground schedule state is only described in terms of resource/2 and task/5, abstract states might also use *derived* predicates such as totalNumberOfTasks/1 which comprise the BK. *SmartGantt* uses background knowledge for the induction of logical trees to represent *Q*-functions, and to evaluate if a ground schedule state belongs to a certain abstract state. Hence, all predicates present in the knowledge base can be used recursively to generate complex combinations of tests involving abstract states and repair actions. This is a powerful feature of the inductive logic programming approach which is used here to keep simple test definitions that are derived in terms of other relationships which are true in the schedule state that is being evaluated. As an example, consider the test

**T1**: `test(time_Prog_In_Res(Res1,Time1), time_Prog_In_Res(Res2,Time2), Time1>Time2)`.which can be derived using the available background knowledge. Basically, this test is true when the total processing time of the scheduled tasks in the resource Res1 is greater than the total time of scheduled tasks in Res2 (Note that Res1 and Res2 might be any pair of different resources in the system). The truth value of the test T1 cannot be determined directly, because it does not exist in the set of facts that determine the schedule state, but is based on a Prolog rule "`time_Prog_In_Res/2`" defined in the BK. This rule needs to obtain the list of tasks programmed in a resource – i.e. `resource(Res,_,Tasks)`, which exists in the schedule state definition – and thereafter calculates the required time using the `sum_Times/2` predicate, which in turn takes as an argument a list of tasks (Tasks variable). If the recursive execution of these predicates obtains a pair of values Time1 and Time2 that makes true Time1 > Time2 (and consequently, a pair of resources Res1 and Res2) then the test T1 can be used as a candidate to carry out the node splitting process (if the test is done as part of the regression tree induction), or take part in the derived Prolog rules which encode the rescheduling knowledge (if the test is used as part of the Abstract State checking process or when a *Q*-value is retrieved). Furthermore, *BK* also might specify constraints such as

BK: $\forall$`XY (precedes(X,Y)` $\rightarrow$ `X/=Y)`,that states that if task `X` precedes task `Y`, they should be different objects.

The use of BK can cover different types of information for a given domain, like experts or users having general domain knowledge, first principles, value function information, partial knowledge about the application of a repair operator, similarities between

**Table 1**
An example problem formulation (Musier & Evans, 1989).

| | A | B | C | D |
|---|---|---|---|---|
| Processing rate (lb/h) | | | | |
| Extruder #0 | 100 | 200 | – | – |
| Extruder #1 | 150 | – | 150 | 300 |
| Extruder #2 | 100 | 150 | 100 | 200 |

| Previous operation | Next operation | | | |
|---|---|---|---|---|
| | A | B | C | D |
| Cleanout requirements (h/cleanout) | | | | |
| A | 0 | 0 | 0 | 2 |
| B | 1 | 0 | 1 | 1 |
| C | 0 | 1 | 0 | 0 |
| D | 0 | 2 | 0 | 0 |

different states, or new relations derived from the relational states. The main advantage of resorting to BK in *SmartGantt* is that different relationships can be inferred between domain objects (tasks and resources), which affects decisively the steepness of a learning curve (Shapiro et al., 2001). In addition, due to the presence of the inductive logic programming component, it is necessary to provide the learning system with some kind of declarative bias specification through types and modes. This technique is employed to reduce the hypotheses search space and mostly focusing on the most relevant feature for goal achievement. Finally, in the definition of each predicate that can take part in a logical query, the type of the arguments should be established which in turn will restrict the types of queries that can be generated.

## 5. *SmartGantt* prototype

The prototype application has been implemented in Visual Basic.NET 2005 Development Framework 2.0 SP2 and SWI Prolog 5.6.61 running under Windows Vista. **TILDE** and **TG** modules from The **ACE Datamining System** developed by the Machine Learning group at the University of Leuven have been used. Fig. 5 depicts *SmartGantt* graphical user interface. The prototype admits two main modes of use: *training* and *consult*. The first task to be performed is training, in which the system learns to repair schedules through simulated transitions of schedule states, and encodes the resulting experience in the *Q*-function so that SmartGantt can apply this rescheduling knowledge in the Consult mode. Before starting the training phase, by means of the graphical interface, the user must define the simulation and training parameters, which are related to the initial schedule conditions, learning parameters and goal state definition. The latter specifies the objective for repairing a schedule selected from an option list including:

- *Tardiness improvement:* Total Tardiness for the repaired schedule should be smaller than the one for the initial schedule. This goal prioritizes the efficiency of the resulting schedule. The reward is assigned as the difference between Total Tardiness present before and after the application of a sequence of repair operators.
- *Stability*: The learning agent tries to minimize the number of changes made to the initial schedule, and rewards are assigned accordingly using:

$$r = \begin{cases} \dfrac{n_0 - n_c}{n_0} & \text{if goals}\,(s_t) = \text{true} \\ 0 & \text{if goals}\,(s_t) = \text{false} \end{cases} \tag{2}$$

where $n_0$ is the total number of tasks, and $n_c$ is the number of tasks that have changed their position with respect to the initial schedule. The Stability goal tries to minimize the operational impact of changes made to the original schedule. This goal is critical because by changing task start times some transportation costs may increase, e.g. if the tools or materials are delivered earlier than required (Pfeiffer et al., 2007). Also, an important issue is that if a rescheduling operation only optimizes an efficiency measure, e.g. Total Tardiness, typically it will generate schedules that are often significantly different from the previous ones. This means that many, if not all, of the previously scheduled tasks that have not begun processing can have their start times changed. This effect is troublesome in practice, especially in the common situation where the process being scheduled uses material that must be delivered from external sources. Such situation can produce severe disturbances in an assembly operation, where the material planner would be required to constantly expedite orders and potentially hold excess inventory for a long period of time trying to support each new schedule being generated, which in most cases is simply unacceptable (Rangsaritratsamee et al., 2004).
- *Balancing*: Tries to trade off tardiness with the number of changes made to the original schedule (Stability goal) in order to achieve the repair goal. Thus, rewards are defined as follows:

$$r = \frac{T_{Initial} - T_{Final}}{N} \tag{3}$$

where $N$ is the number of required steps for achieving the goal state and $T_{Initial}$ and $T_{Final}$ are the tardiness for the initial schedule and the repaired one, respectively.

In the above repair objective definitions, each option means to change the conditions that define the goal state and the way in which the reward is assigned to the applied repair operators, prioritizing either efficiency or stability, or a balance between them. At the beginning of each training episode, the prototype generates randomly a set of scheduled orders with their corresponding attributes, bounded over the allowable ranges for them which are defined interactively.

To generate the initial schedule state $s_0$, these orders are randomly assigned to one of the available resources. Later on, the attributes of the order to be inserted – without increasing the Total Tardiness in the initial schedule – are generated, and the corresponding tasks are arbitrarily scheduled in the available resources. The learning episode progresses by applying a sequence of repair operators until the goal state is reached. Based on a suitable initialization of the *Q*-function, the RRL algorithm starts by trial-and-error learning with simulated episodes while updating its knowledge base using the standard *Q*-learning algorithm (see Eq. (1)). In each training episode, every encountered state-action pair is stored along with the associated reward. At the end of each episode, when the learning agent has reached the goal state, the *Q*-values of each state-action pair visited are updated using back-propagation. So, the RRL algorithm gives the set of tuples (state, action, *Q*-value) to a relational regression engine, employing this set of examples to update the regression tree that represents the *Q*-function. Then, the algorithm continues executing the next episode in the learning curve. In the generated tree, different nodes are basically Prolog queries. As a result, to find a *Q*-value, a Prolog program based on the tree is built online. After that, when the query (state-action) is executed, this engine will return the desired value, or the best repair operator to be applied for a given schedule state. The estimated *Q*-value will depend on the quality of the generated tree, which in turn relies upon the rules that define the background knowledge.

The relational regression tree contains (in relational format) the repair policy learned from the current and previous learning episodes. The mentioned queries are actually processed by the Prolog wrappers **ConsultBestAction.exe** and **ConsultQ.exe**, which create a transparent interface between the .NET agent and the relational repair policy. A PROLOG.NET library is used to perform the induction process for abstract states. Also, the RRTL module includes the functionality for discretizing continuous variables such as Total Tardiness and Average Tardiness in non-uniform real-valued intervals so as to make the generated rules useful for Prolog wrappers.

In the .NET prototype, different classes are used to model *Agent*, *Environment*, *Actions* and *Policy* concepts. The Prolog files Policy.pl, ActState.pl, ActStateAction.pl and BackgroundKnowledge.pl are modified dynamically in execution time. Finally, the .NET agent is fully equipped to handle disruptions and events such as when an order or task cannot be inserted in the initial schedule. To this aim, the agent may modify order attributes such as date or size so as to insert the order or task with different attributes. The prototype allows the user to interactively revise and accept/reject changes made to order attributes. The prototype can show graphically the evolution of the order/task insertion procedure (and the sequence of repair operators applied from the initial schedule) and learning results for the repair policy.

## 6. Industrial case study

An example problem proposed in Musier and Evans (1989) is considered here to illustrate the use of repair operators for batch plant rescheduling. The plant is made up of 3 semi-continuous extruders that process customer orders for four products. Each extruder has distinctive features, so that not all the extruders can process all products. Additionally, processing rates depend on both the extruder and the product being processed. For extruders, set-up times required for resource cleaning have also been provided based on the precedence relationship between product types. Processing rates and cleanout requirements are detailed in Table 1. Order attributes correspond to product type, due date and size.

In this section, the example is used to illustrate concepts like relational definition of schedule states and repair operators, global and focal (local) variables used in the relational model, and the overall process of repairing a schedule bearing in mind different objectives in the goal state, namely stability, efficiency, or a mix between the two, depending on the user's preference. For example, if the disruptive event is the arrival of a new order and the selected goal is *stability*, the aim is not to increase the Total Tardiness and produce the lowest number of changes to the initial schedule when a new/rush order is inserted. Also, a delay in the arrival of raw materials/machine breakdown is considered as a disturbance. In all cases, the schedule situation in the training mode before the sequence of repair operations is applied has been randomly generated by either: (i) arrival of an order with given attributes that should be inserted in a randomly generated schedule state, or a delay in the arrival of raw materials for a given order, and (ii) the arriving order attributes or delay/breakdown time are also randomly defined. This way of generating both the initial schedule and the new/rush order or delay/breakdown exposes the agent to totally different initial situations while learning a repair policy which is able to deal successfully with shop-floor uncertainty and dynamics.

**Table 2**
Global and focal variables in the prototype.

| Name | Description |
|---|---|
| TotalTardiness [h] | **Global variable**. Sum over all tardiness of each task |
| MaxTardiness [h] | **Global variable**. Maximum tardiness of the schedule |
| AvgTardiness [h] | **Global variable**. Total Tardiness divided the number of tasks |
| TotalWIP [lb] | **Global variable**. Total size for all the orders in the schedule |
| TardinessRatio | **Global variable**. Sum over all orders in the schedule of the ratio between tardiness of the order and its lead time |
| InventoryRatio | **Global variable**. Sum over all orders in the schedule of the ratio between processing time of the order and its lead time |
| ResNLoad [%] | **Global variable**. Utilization ratio for resource #*N* in the schedule |
| ResNTardiness [h] | **Global variable**. Tardiness of resource #*N* in the schedule |
| TotalCleanoutTime | **Global variable**. Time spent in cleanout operations |
| FocalTardiness [h] | **Focal variable**. Tardiness associated to the focal task (order) |
| ProductType | **Focal variable**. Product associated to focal task |
| LeftTardiness [h] | **Focal variable**. Sum over all tardiness of each order which is programmed before the focal |
| RightTardiness [h] | **Focal variable**. Sum over all tardiness of each order which is programmed after the focal |
| MaterialArriving [h] | **Focal variable**. Amount of hours in which the raw materials will arrive. It only applies if the disruptive event is "Shortage or delay in the arrival of raw materials" |
| BreakdownInMachine [#] | **Focal variable**. ID of the machine affected by breakdown. It only applies if the disruptive event is "Machine Breakdown" |
| BreakdownTimeStart [h] | **Focal variable**. Time (in h) at which the breakdown start. It only applies if the disruptive event is "Machine Breakdown" |
| BreakdownTotalTime [h] | **Focal variable**. Length of the breakdown, in hours. It only applies if the disruptive event is "Machine Breakdown" |
| FocalRSwappability | **Focal variable**. Binary variable to indicate if it is feasible to swap the focal task with one to the right in the same extruder |
| FocalLSwappability | **Focal variable**. Binary variable to indicate if it is feasible to swap the focal task with one to the left in the same extruder |
| FocalAltRSwappability | **Focal variable**. Binary variable to indicate if it is feasible to swap the focal task with one to the right in a different extruder |
| FocalAltLSwappability | **Focal variable**. Binary variable to indicate if it is feasible to swap the focal task with one to the left in a different extruder |

The initial schedule is generated in terms of several parameters, which can be changed using the graphical interface of the prototype, such as number of orders, order composition (product types), order sizes and due dates. The focal and global variables used in this example are detailed in Table 2, combined with a relational representation of the schedule state that has been discussed previously. Customer orders used in Consult mode to repair a schedule which has been affected by the occurrence of a disruptive event are shown in Table 3. In each training episode, a random schedule state for a certain number of initial orders is generated. The number and general features of the orders can be different between each training episode, which is determinated by the system having into account the parameter values established by the user. Then a random insertion is attempted for the new order (whose attributes are also randomly chosen), which in turn serves as the focal point for defining repair operators thereafter, which may vary

**Table 3**
Data for the initial orders for the example.

| Order # | Product | Size [lb] | DD [h] |
|---|---|---|---|
| 1 | C | 619 | 3 |
| 2 | C | 559 | 18 |
| 3 | A | 570 | 8 |
| 4 | C | 541 | 17 |
| 5 | B | 694 | 17 |
| 6 | C | 222 | 1 |
| 7 | D | 237 | 18 |
| 8 | D | 695 | 16 |
| 9 | B | 662 | 10 |
| 10 | B | 695 | 18 |
| 11 | C | 575 | 14 |
| 12 | B | 191 | 4 |
| 13 | A | 932 | 13 |

according to the chosen repair goal. For example, the goal state for the repaired schedule can be stated in terms of the Total Tardiness (TT) as follows:

"the order #13 must be inserted without increasing the TT present in the current schedule."

Training and consult times for each disruptive event are given in Table 4. The training process is carried out only once. Thereafter, the system can be used to repair schedules in Consult mode; such schedules can be different from those generated automatically by the system during training.

To illustrate the advantages of RRL in real-time rescheduling, once training has been completed four specific situations have been considered. In the first case, the 12 orders detailed in Table 3 are already scheduled in the plant and a new order #13 must be inserted so that the Total Tardiness (TT) in the schedule is not increased. Only 6 repair steps are required, on average, to insert the order #13 (regardless of the number of orders previously scheduled). Fig. 6 provides an example of applying the optimal sequence of repair operators from the schedule in Fig. 6(a) using the Consult mode of *SmartGantt*, and choosing stability as the repair goal. Before the order #13 has been inserted, the Total Tardiness (TT) is 26.27 h. Once the arriving order (in white) has been inserted, the Total Tardiness has been increased to 39.79 h; orange tasks are used to indicate cleaning operations. Based on the learned repair policy, a `RightSwap` operator should be applied, which gives rise to the schedule in Fig. 6(b) with a TT = 27.59 h. Finally, by means of a `LeftSwap` the goal state is reached with a Total Tardiness of 25.24 h, which is even lower than the TT in the initial schedule before the 13th order was inserted. As can be seen in the repair sequence, the policy tries to swap orders in order to improve the utilization of the first extruder. It is important to note the small number of steps that are required for the rescheduling agent to implement the learned policy for order insertion. A word of caution, though, is that the initial schedule is not necessarily an optimized one, so there is enough room for improving tardiness.

In the second scenario, there exists the same 12 orders already scheduled, and a delay of 4 h in the arrival of raw materials is generated for orders of the product type A for orders #3 and #13. These orders must be reprogrammed so as to face with the unplanned event and to meet the chosen rescheduling goal (Stability). After 550 training episodes, only 9 repair steps are required, on average, to move the start times of order #3 and #13 in order to handle the delayed arrival of raw material. The initial schedule is similar to the one given in Fig. 6(a). Before the #13 order has been moved, the Total Tardiness is 26.27 h (Fig. 5a). Fig. 7 provides an example of applying the optimal sequence of repair operators from the initial schedule, using the Consult mode of *SmartGantt*, and pursuing the Stability goal. Based on the learned repair policy, *SmartGantt* applies a `RightMove` operator, trying to delay the start time of Order #13, which gives rise to the schedule in Fig. 7(a) with a TT = 39.90 h. Despite the new start time for order #13 is satisfactory, order #3 is still schedule to begin before the planned arrival of raw material and the tardiness has been incremented. So, the system applies the `DownLeftSwap` operator, trying to change the assigned resource for orders #3 and #13 and at the same time, delay the start time of the former. This operation gives rise to the schedule shown in Fig. 7(b), with a TT = 34.98 h. Although order #3 is now scheduled to start after the arrival of materials, the goal state is not yet reached, because the amount of tardiness present in the schedule is greater than the one for the initial schedule and order #13 begins before the arrival of raw materials. Then the system applies the `DownRightSwap` operator, which changes the assigned resources for the orders #4 and #13, generating the schedule in Fig. 7(c) with TT = 42.89 h. At this point, the user must tell the system whether to continue with the repairing process, or accept the current schedule. If the user decides to continue, *SmartGantt* tries to reduce tardiness applying the next sequence of repair operators: `LeftJump` (Fig. 7(d)), `UpRightSwap` (Fig. 7(e)), `DownLeftJump` (Fig. 7(f)), `LeftMove` (Fig. 7(g)) and `DownRightSwap` (Fig. 7(h)). The goal state is then reached after applying an `UpLeftJump` operator, with a Total Tardiness of 24.67 h, which is even lower than the TT in the initial schedule.

The relational heuristic applied from a given initial schedule by *SmartGantt* by means of Prolog rules learned via state transition simulation (which can be different for other schedule initial conditions and configurations) shows vividly how the use of a relational-deictic representation allows the policy becoming meaningful for the operator, and can be synthesized, for example, through the next statement:

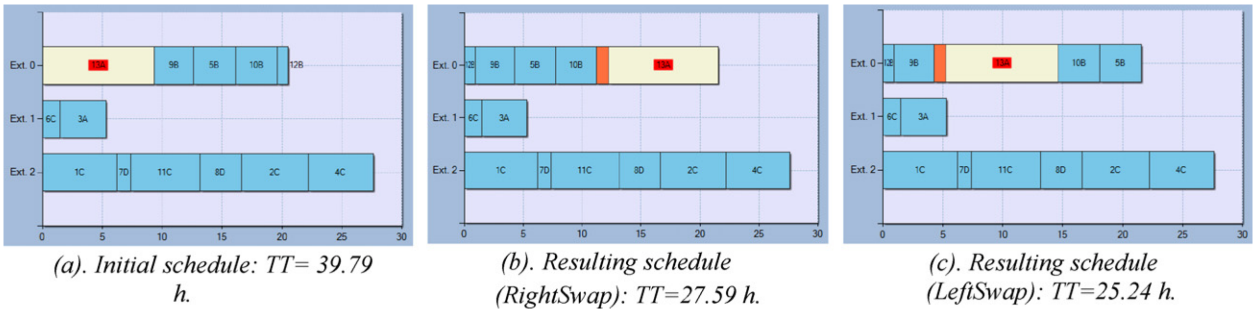
"If the start time of the affected orders is lower than the new date for raw material availability, try to move the orders to the right in the same resource. If the goal is not reached, try to move the order to an alternative resource so as the produce a delayed start time of the order, but not changes should be made to the relative position of the other scheduled orders. If the affected task is the last to be processed in the resource, then perform swap operations with orders placed in an alternative machine which has more tasks scheduled than the ones for the first resource. If the rescheduling goal has not yet reached, then try a more complex operator to produce a higher delay in the start time of the orders but involving a swap operation with a task scheduled in an alternative resource".

In the third situation, let's assume that the orders described previously are already scheduled in the plant, and an arriving rush order must be inserted so as to meet the repairing goal (Balancing, Stability or Tardiness Improvement) and an extra condition regarding the due date of the Rush Order must be enforced. Training process has been performed in the same manner as for the arrival of a new order event, but the due date for the order #13 has been changed to **8** to represent a more stringent delivery condition. For the Stability goal, a near-optimal repair policy is obtained after approximately 470 episodes. For the other two goals, the learning process tends to stabilize with more training episodes due to the tighter constraints imposed by the repairing goal (e.g. reduce the Total Tardiness, and at the same time respect the Rush Order due date).

Fig. 8 provides a comprehensive picture of the optimal sequence of repair operators from the schedule in Fig. 8(a) when the Stability goal is pursued. Before the order #13 has been scheduled, the Total Tardiness is 26.27 h. Once the arriving rush order (in white) has been inserted, the Total Tardiness has been increased to 44.90 h. Based on the learned repair policy, a `LeftJump` operator is applied, which gives rise to the schedule in Fig. 8(b) with a TT = 41.12 h. Later on, a `RigthSwap` operator is used which decreases tardiness

**Table 4**
CPU times for *Training* and *Consult* mode.

| Disruptive event | Training mode | | | Consult mode | |
|---|---|---|---|---|---|
| | # Episodes | # Steps (Avg.) | Total time [h] | # Steps to repair | Total time [s] |
| Arrival of a new order | 400 | 6 | 1.14 | 3 | 1.07 |
| Delay or shortage in the arrival of raw materials | 550 | 9 | 1.50 | 9 | 2.34 |
| Rush order | 470 | 7 | 1.35 | 4 | 1.20 |
| Machine breakdown | 700 | 10 | 2.34 | 4 | 1.78 |

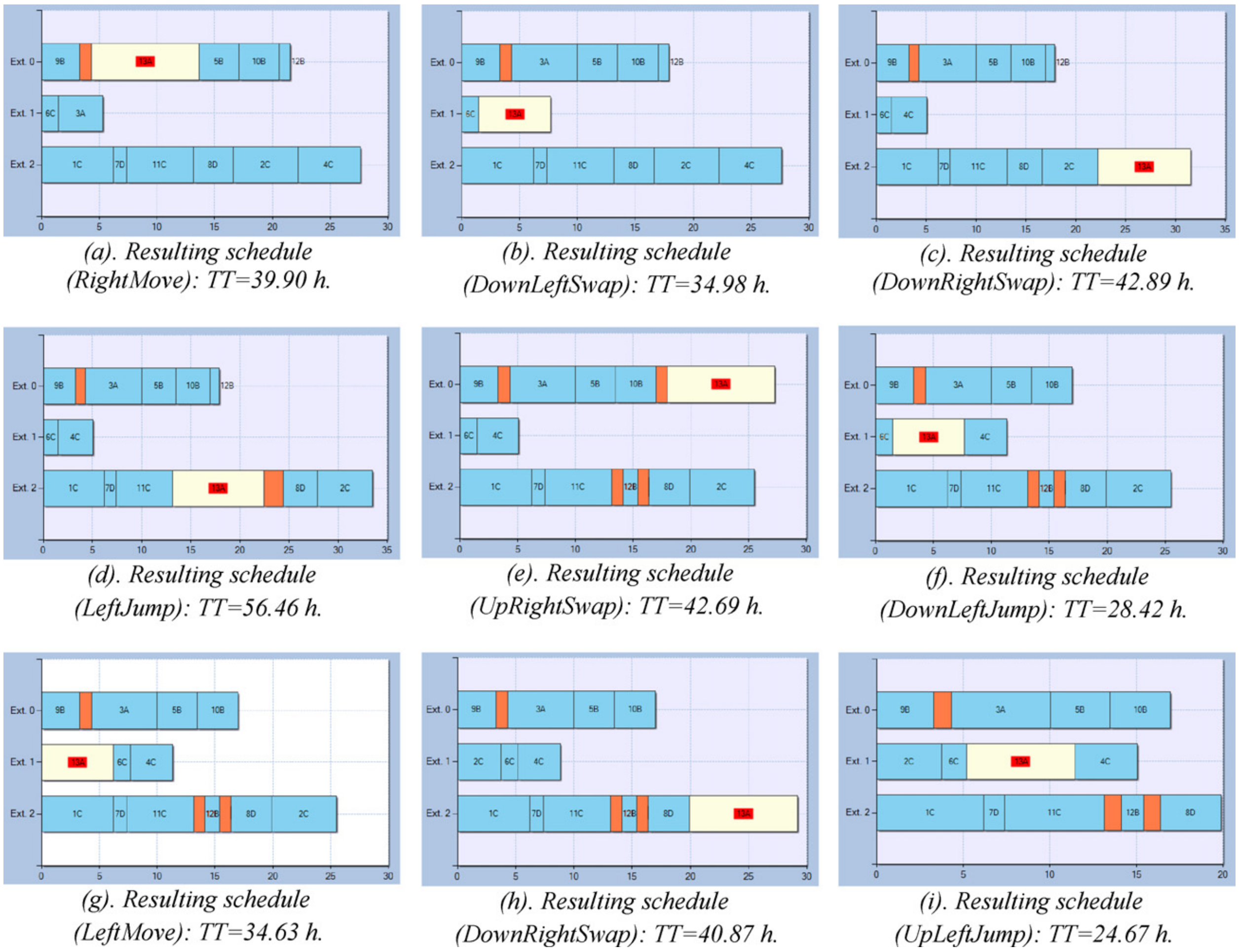


*(a). Initial schedule: TT= 39.79 h.* *(b). Resulting schedule (RightSwap): TT=27.59 h.* *(c). Resulting schedule (LeftSwap): TT=25.24 h.*

**Fig. 6.** Example of applying the optimal sequence of repair operators for the disruptive event *order arrival*.

to 32.59 h (see Fig. 8(c)). Since the goal state is not yet reached, the system uses a `DownRightJump` operator to move the focal task to an alternative resource, as can be seen in Fig. 8(d). Finally, the goal state is reached by resorting to an `UpLeftJump` operator with a Total Tardiness of 22.55 h, which is even lower than the TT in the initial schedule. The heuristic for repair in this particular case can be phrased as follows:

"to insert a rush order, try to move the focal task to the left in the same resource. If the goal is not reached, try to reduce tardiness swapping the position of the focal task with another task



*(a). Resulting schedule (RightMove): TT=39.90 h.* *(b). Resulting schedule (DownLeftSwap): TT=34.98 h.* *(c). Resulting schedule (DownRightSwap): TT=42.89 h.* *(d). Resulting schedule (LeftJump): TT=56.46 h.* *(e). Resulting schedule (UpRightSwap): TT=42.69 h.* *(f). Resulting schedule (DownLeftJump): TT=28.42 h.* *(g). Resulting schedule (LeftMove): TT=34.63 h.* *(h). Resulting schedule (DownRightSwap): TT=40.87 h.* *(i). Resulting schedule (UpLeftJump): TT=24.67 h.*

**Fig. 7.** Example of applying the optimal sequence of repair operators for the disruptive event *delay in raw material*.

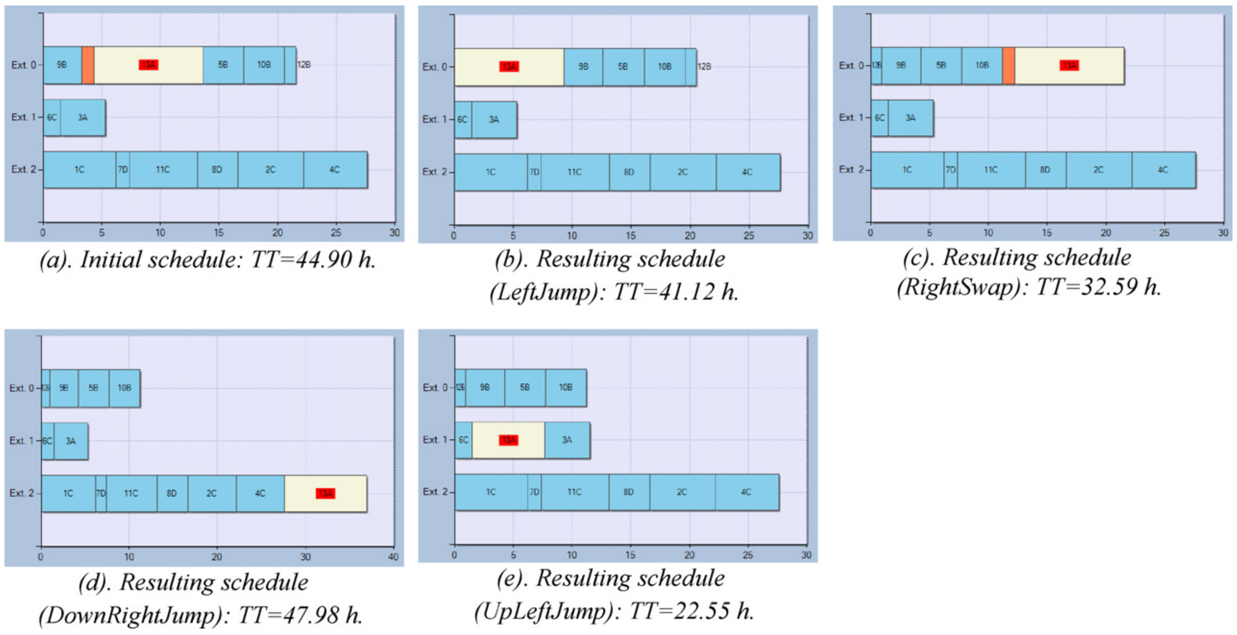


**Fig. 8.** Example of applying the optimal sequence of repair operators for the disruptive event *rush order*.

located on its right and then use a more complex combination of operators to place the focal task in an alternative resource, thus reducing its start time."

The last disruptive event that has been considered is an extruder breakdown. For simulation purposes, is assumed that the same set of orders that were considered previously is already scheduled in the plant. Then, an unexpected breakdown for a limited amount of time is generated randomly for a given extruder. Then, the system must select a task to take as focal point, and then the rescheduling goal is pursued. For this event, in a high number of episodes *SmartGantt* must relax the conditions stated for achieving the goal state, particularly in those cases where the user selects Tardiness Improvement or Balancing as goals. After approximately 700 training episodes, on average 10 repair steps are required to perform the rescheduling task for the Stability goal. For the other two goals, the process converges only after 900 training episodes, possibly due to the very stringent conditions established by the nature of repair schedules that are expected and the number of operations needed to reach the chosen goal state.

Fig. 9 provides an example of applying the optimal sequence of repair operators starting from the schedule in Fig. 9(a) and having chosen the Stability goal. Before the resource breakdown event occurs, the Total Tardiness is 39.80 h. Then, a breakdown of 3 h in the extruder #2 is generated, which gives rise to the schedule in Fig. 9(a) with a TT = 58.20 h. Based on the learned repair policy, *SmartGantt* select as focal the order #13, and applies the `DownRightJump` operator, trying to reduce the Total Tardiness, which is decreased to 44.67 h, as can be seen in Fig. 9(b). Then,

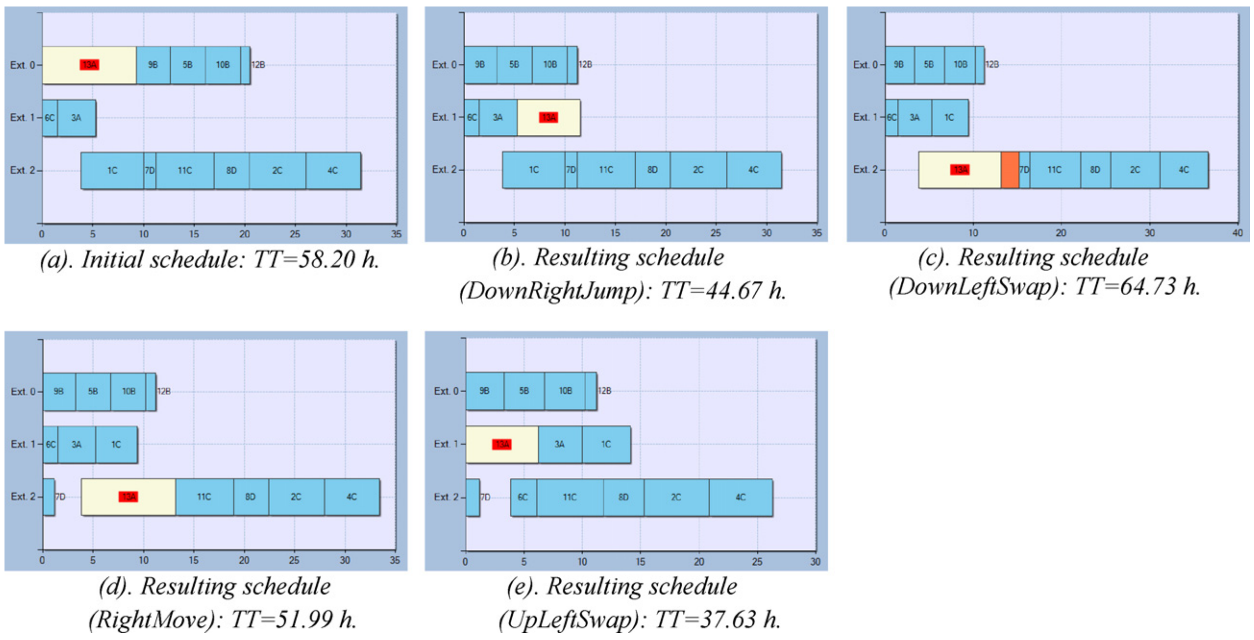
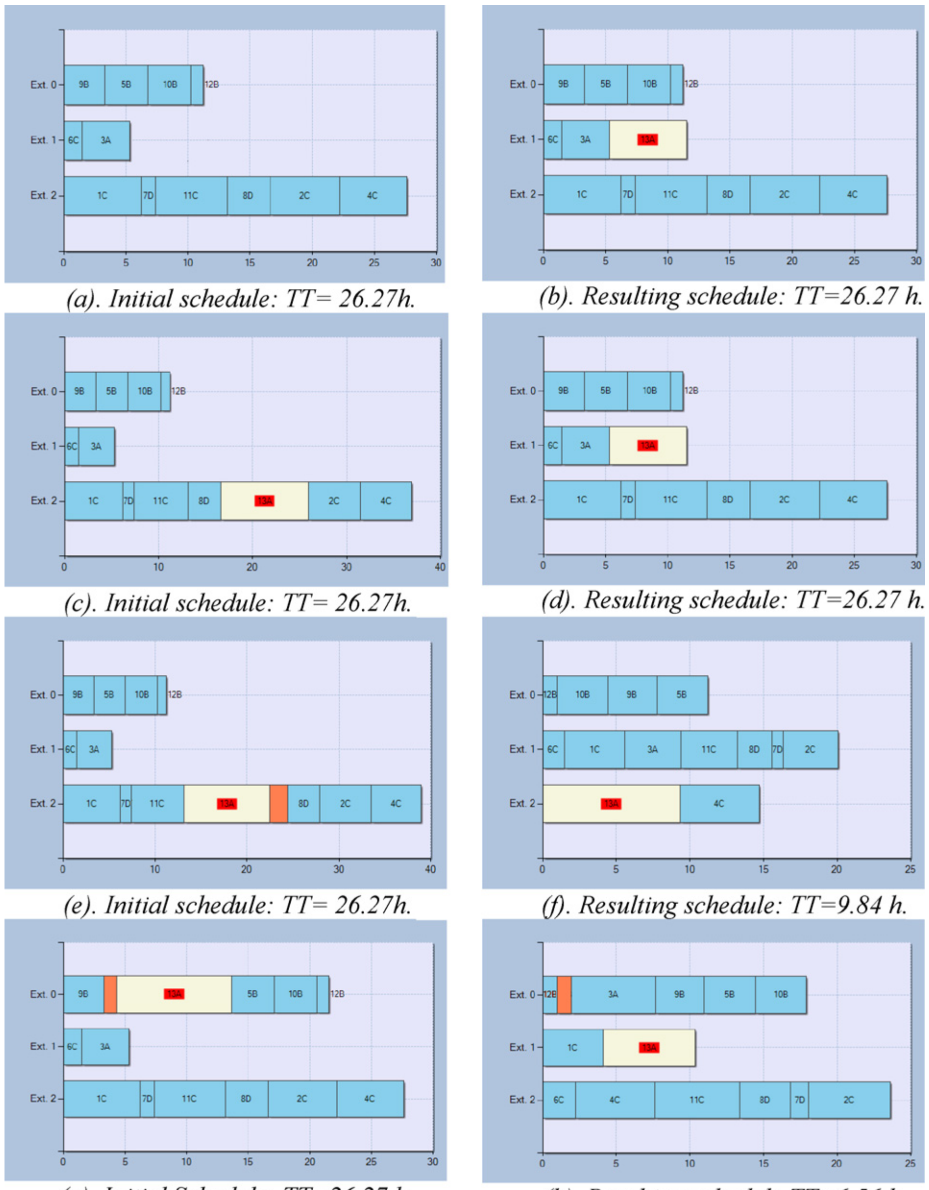


**Fig. 9.** Example of applying the optimal sequence of repair operators for the disruptive event *extruder breakdown*.



(a). Initial schedule: TT= 26.27h.
(b). Resulting schedule: TT=26.27 h.
(c). Initial schedule: TT= 26.27h.
(d). Resulting schedule: TT=26.27 h.
(e). Initial schedule: TT= 26.27h.
(f). Resulting schedule: TT=9.84 h.
(g). Initial Schedule: TT=26.27 h.
(h). Resulting schedule TT=6.56 h.

**Fig. 10.** Example of applying rescheduling heuristics. (a and b) 1Optimal; (c and d) 2Optimal; (e and f) 3Optimal; (g and h) 4Optimal.

*SmartGantt* applies a `DownLeftSwap` operator, increasing TT to 64.73 h. Since the goal state has not been yet reached, the rescheduling system applies the `RightMove` operator – which reduces tardiness and allows that order #7 be placed in the first position of the resource – and finally the repair goal is reached using the `UpLeftSwap` operator, which reduces the TT to 37.63 h, as it is shown in Fig. 9(e). The last operator reduces tardiness placing a smallest order (#4) in the affected resource, and take advantage of the underutilized resource. It is important to highlight the usefulness of repair operators in learning of the repair policy for a resource breakdown, which allows the system to exploit small idle times left in all resources.

It is noteworthy that once the *Q*-function is known, reactive scheduling is straightforward. Moreover, for approximating this decision-making function the only thing we need is to simulate state transitions between abstract schedule states. The applicability of a comprehensive policy for repair schedules on the fly is a novel approach that facilitates designing the human–agent–machine interface.

## 7. Benchmarking

To perform a comparison of the repair policy learned by SmartGantt, heuristics **1Optimal**, **2Optimal**, **3Optimal** and **4Optimal** adapted from Musier and Evans (1989) have been implemented. In this way, heuristics consist of making successive position changes of each of the scheduled tasks following predefined rules, with the aim of reducing the Total Tardiness in the system. The 1Optimal operator is the simplest one, because it places the focal order at the end of resource with minimum Total Tardiness. Using the 2Optimal operator the order is inserted between two existing orders so as to generate the least possible tardiness increase. Both operators do not make changes in the previous position of the scheduled tasks. The 3Optimal operator involves testing the schedule tardiness after changing the position of all orders taken one by one. So, 3Optimal starts with the order labeled as #1, and proceeds to reallocate such order after all the remaining orders. This operation generates several new possible schedules, of which the system stores the one that has lower Total Tardiness. Such schedule is used

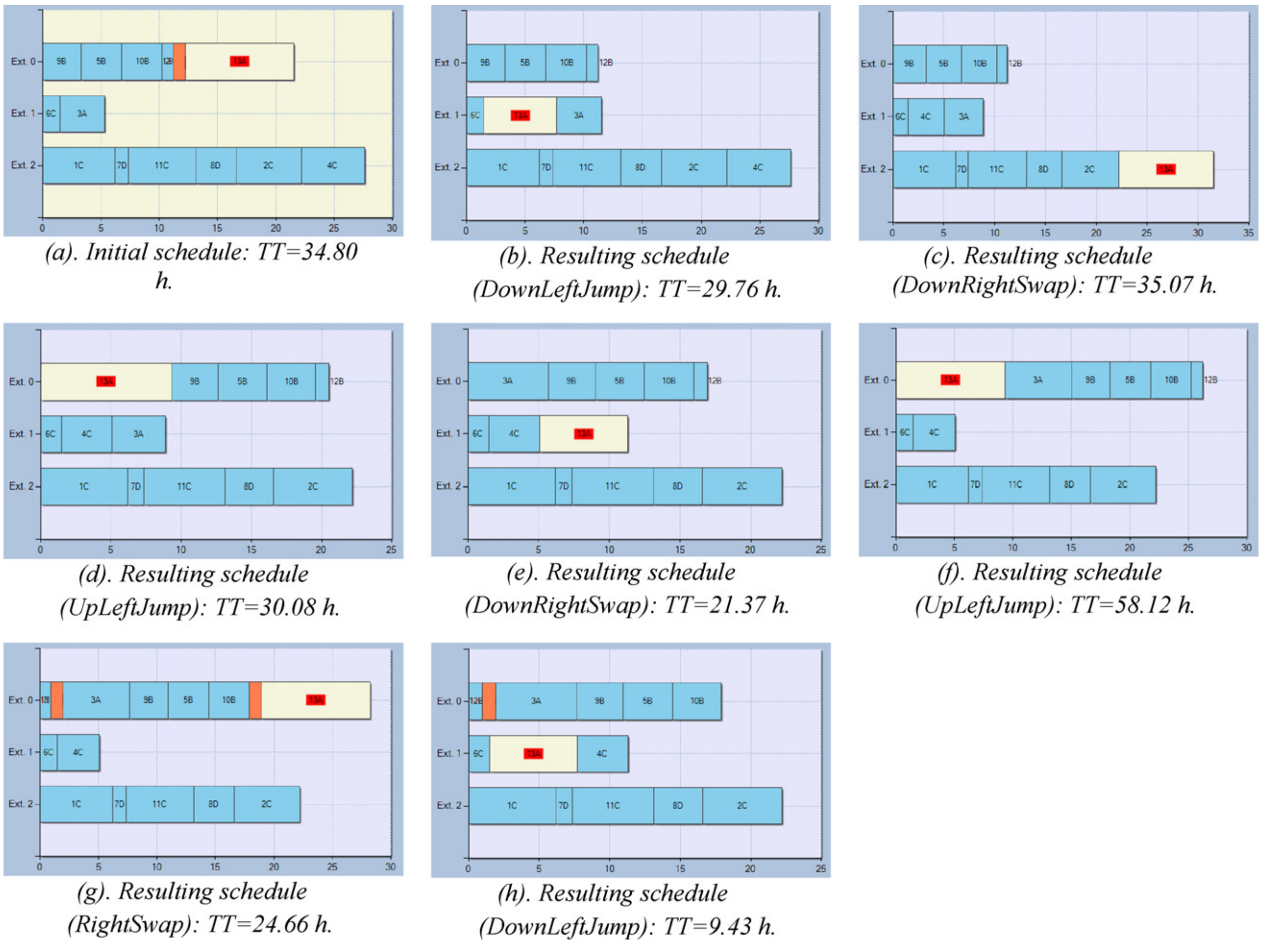

(a). Initial schedule: TT=34.80 h.
(b). Resulting schedule (DownLeftJump): TT=29.76 h.
(c). Resulting schedule (DownRightSwap): TT=35.07 h.
(d). Resulting schedule (UpLeftJump): TT=30.08 h.
(e). Resulting schedule (DownRightSwap): TT=21.37 h.
(f). Resulting schedule (UpLeftJump): TT=58.12 h.
(g). Resulting schedule (RightSwap): TT=24.66 h.
(h). Resulting schedule (DownLeftJump): TT=9.43 h.

**Fig. 11.** Optimal sequence of repair operators for the Tardiness Improvement goal.

as the basis for performing the same operation again but taking the order #2 recursively, until no more orders are left to consider. The resulting schedule is named 3Optimal. Finally, 4Optimal is the more complex operator because it proceeds in the same manner of 3Optimal, but instead of taking the orders one at a time and test possible position changes regarding the others, it takes orders in pairs and performs a swap operation between them. Therefore, 4Optimal starts with the order labeled as #1, and proceeds in a sequential way to swap this order with order #2, #3, #4, etc. The schedule with minimum tardiness is stored, proceeding in the same manner as in operator 3Optimal. Then, the resulting schedule is called 4Optimal. It is noteworthy that by definition, a 4Optimal schedule is globally optimal, since it involves finding the best repositioning of all orders into all possible schedule positions.

After implementing the aforementioned heuristics, a general training for the event *order arrival* has been carried out by considering all the available repair goals. Then, an initial schedule with the 12 tasks already scheduled has been generated, as well as a new order to be inserted (#13), as it is detailed in Table 3. Fig. 10 shows the initial and resulting schedule obtained after the application of each repair operator. Results obtained with the repair policy learned by SmartGantt for each goal are shown in Figs. 11–14.

The first heuristic repair rule that has been applied was the 1Optimal (Fig. 10(a) and (b)), then the 2Optimal (Fig. 10(c) and (d)) schedule repair strategy has been applied, and later on the 3Optimal (Fig. 10(e) and (f)) and eventually the 4Optimal was used (Fig. 10(g) and (h)). Finally, the repair policy learned through intensive simulation was applied, considering as repair goals: Tardiness Improvement (Fig. 11), Stability (Fig. 12) and Balancing (Fig. 13).

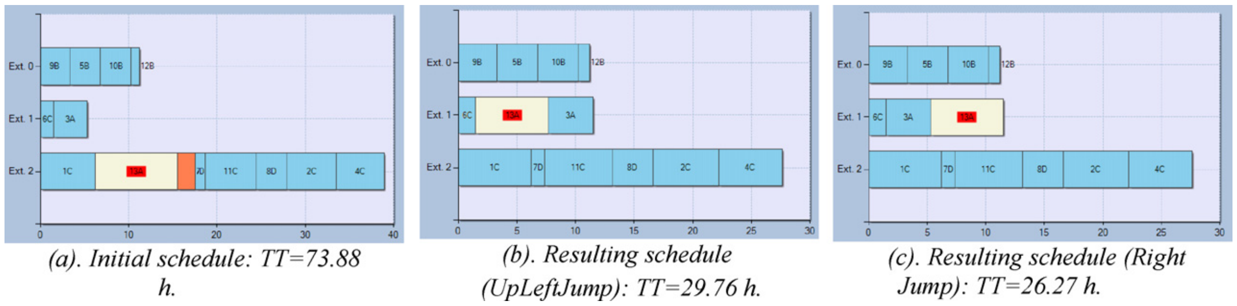
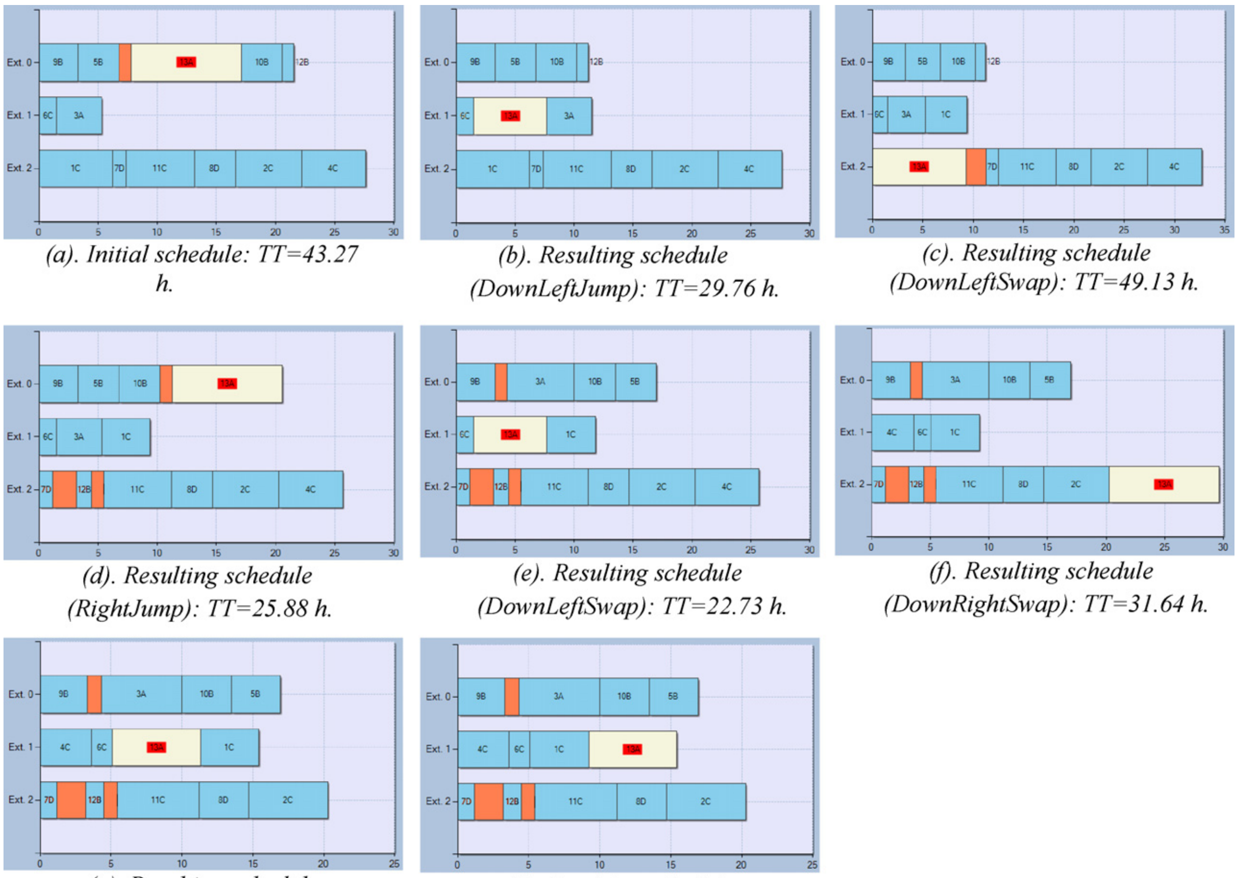

(a). Initial schedule: TT=73.88 h.
(b). Resulting schedule (UpLeftJump): TT=29.76 h.
(c). Resulting schedule (Right Jump): TT=26.27 h.

**Fig. 12.** Optimal sequence of repair operators for the Stability goal.



*(a). Initial schedule: TT=43.27 h.*

*(b). Resulting schedule (DownLeftJump): TT=29.76 h.*

*(c). Resulting schedule (DownLeftSwap): TT=49.13 h.*

*(d). Resulting schedule (RightJump): TT=25.88 h.*

*(e). Resulting schedule (DownLeftSwap): TT=22.73 h.*

*(f). Resulting schedule (DownRightSwap): TT=31.64 h.*

*(g). Resulting schedule (UpLeftJump): TT=21.26 h.*

*(h). Resulting schedule (RightSwap): TT=17.47 h.*

**Fig. 13.** Optimal sequence of repair operators for the Balancing goal.

In all cases, the initial schedule consists of the same set of programmed tasks; the only difference is the initial position of the arriving task, which depends on the particular heuristic used, the 1Optimal, 2Optimal, 3Optimal and 4Optimal operators, and the policy learned by SmartGantt for the specific repair goal. Subsequently, comparisons of the rewards obtained by each of the repair strategies used were performed.

Fig. 11 highlights that the results obtained from the application of the heuristics previously mentioned and the policy learned by SmartGantt for Tardiness Improvement repair goal. The greatest reduction of Total Tardiness is obtained by the operator 4Optimal (19.71 h), followed by SmartGantt (16.85 h). As can be seen in this figure, 3Optimal is able to reduce the Total Tardiness in 16.43 h. 1Optimal and 2Optimal operators cannot reduce tardiness. It is worth noting that although the tardiness reduction obtained by SmartGantt is less than the one obtained by the 4Optimal (which is also the global optimum) it shows a better performance compared with the 3Optimal heuristics. Another advantage of the present approach relies on the fact that the learned policy is expressed in a Prolog program which is re-built on-line and can be used in real time in a straightforward way. On the contrary, using 3Optimal and 4Optimal heuristics require performing all the necessary schedule changes each time a given heuristics is applied. Such task can be computationally very demanding depending on the number of tasks in the initial schedule. In this particular case, to apply the 4Optimal strategy requires to evaluate 156 different schedules each time. Instead, the repair policy can be used for reactive rescheduling by applying only 7 rules. Moreover, it is possible to nearly match the performance of the 4Optimal heuristic using the learned repair policy.

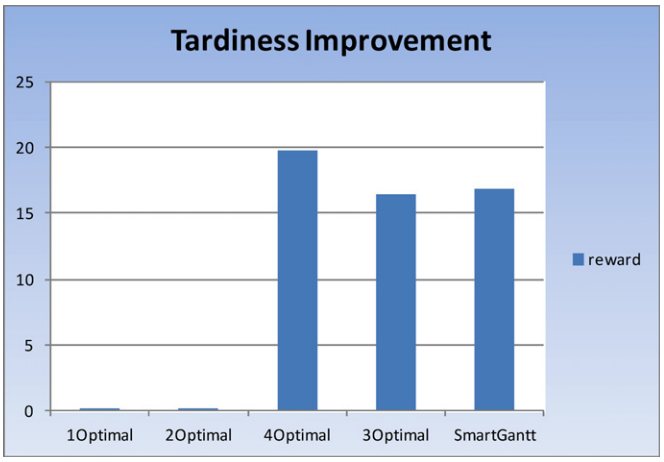


**Fig. 14.** Reward comparison for the Tardiness Improvement goal.

Fig. 15 shows the results obtained for the Stability repair goal. In this case, the operators 3Optimal and 4Optimal get a reward of just 0.15, whilst the learned repair policy gets a reward of 1. This means that the policy is able to insert the new order, without producing an increment in the level of tardiness present in the initial schedule, as can be seen in Fig. 12(c). Moreover, this insertion is made without a major impact in terms of position changes to the tasks

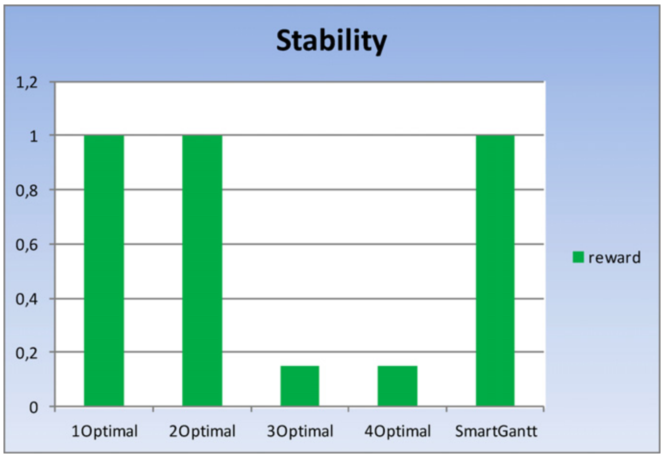


**Fig. 15.** Reward comparison for the Stability goal.

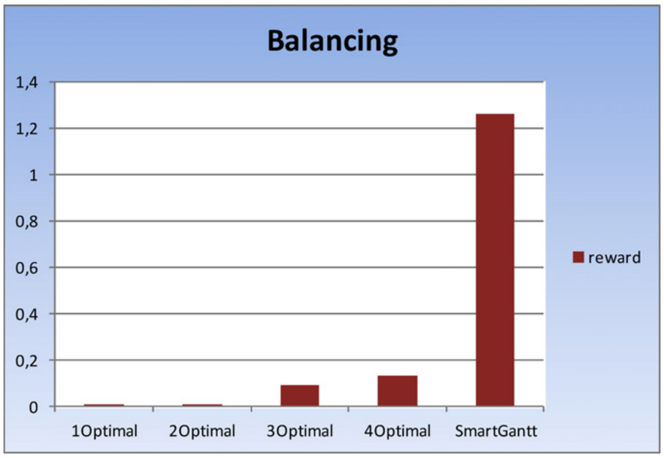


**Fig. 16.** Reward comparison for the Balancing goal.

in the original schedule (actually all tasks has maintained its original position in the repaired schedule). Comparatively, the 4Optimal and 3Optimal operators are able to carry out order insertion successfully, but generate a very different schedule from the initial one. On the other hand, despite resorting to the 1Optimal and 2Optimal have not changed task positions in most cases it is impossible to insert the order using 1Optimal or 2Optimal without give rising to an increment in the Total Tardiness.

In Fig. 16 the results obtained for the Balancing repair goal are given. For this goal, the 4Optimal operator gets a reward of 0.13, whereas for the 3Optimal is just 0.09. 1Optimal and 2Optimal operators get no reward at all. The repair policy applied by *SmartGantt* provides results given in Fig. 13 and gets instead a reward of 1.26, thereby achieving the best trade-off between reducing tardiness and the number of tasks that must change their positions to reach the goal state.

## 8. Concluding remarks

A novel approach for simulation-based development of a relational policy for automatic repairing schedules in real time using reinforcement learning has been presented along with a prototype implementation. The policy allows generating a sequence of deictic (local) repair operators to achieve different rescheduling goals to handle abnormal and unplanned events, such as inserting an arriving order with minimum tardiness or responding to a raw material delay/shortage. Representing schedule abstract states using a relational abstraction is a very natural choice to mimic the human ability to deal with rescheduling problems, where relations between objects and focal points for defining repair strategies are typically used. Moreover, using relational modeling for learning from simulated examples is a very appealing approach to compile a vast amount of knowledge about rescheduling policies, where different types of abnormal events (order insertion, resource failure, rush orders, reprocessing operations, etc.) can be generated separately through intensive simulation and then compiled in the relational regression tree for the repair policy, regardless of the event used to generate the atomic schedule states.

## Acknowledgments

The authors would like to thanks Prof. Hendrik Blockeel and "Declaratieve Talen en Artificiele Intelligentie" (DTAI, http://dtai.cs.kuleuven.be/) Research Group at the *Katholieke Universiteit Leuven*, Belgium, for kindly providing the ACE Data Mining tool used in part of this work.

## References

Adhitya, A., Srinivasan, R., & Karimi, I. A. (2007). Heuristic rescheduling of crude oil operations to manage abnormal supply chain events. *AIChE Journal*, *53*, 397.

Blockeel, H., & De Raedt, L. (1998). Top-down induction of first order logical decision trees. *Artificial Intelligence*, *101*, 285.

Blockeel, H., De Raedt, L., Jacobs, N., & Demoen, B. (1999). Scaling up inductive logic programming by learning from interpretations. *Data Mining and Knowledge Discovery*, *3*, 59.

Chapman, D., & Kaelbling, L. P. (1991). *Input generalization in delayed reinforcement learning: An algorithm and performance comparison. Proceedings of the 12th international joint conference on Artificial intelligence (IJCAI'91)* San Francisco, USA: Morgan Kaufmann Publishers Inc. (pp. 726–731)

Croonenborghs, T. (2009). *Model-assisted approaches to relational reinforcement learning*. Ph.D. dissertation, Dept. of C. Sc., K. U. Leuven, Leuven, Belgium.

De Raedt, L. (2008). *Logical and relational learning*. Berlin: Springer-Verlag.

Driessens, K., Ramon, J., & Blockeel, H. (2001). Speeding up relational reinforcement learning through the use of an incremental first order decision tree learner. In L. De Raedt, & P. Flach (Eds.), *Thirteenth European conference on machine learning* (p. 97). Heidelberg: Springer.

Driessens, K., & Ramon, J. (2003). *Relational instance based regression for relational reinforcement learning. 20th International conference on machine learning* Washington: AAAI Press.

Džeroski, S., De Raedt, L., & Driessens, K. (2001). Relational reinforcement learning. *Machine Learning*, *43*, 7.

Henning, G., & Cerdá, J. (2000). Knowledge-based predictive and reactive scheduling in industrial environments. *Computers and Chemical Engineering*, *24*, 2315.

Li, Z., & Ierapetritou, M. (2008). Reactive scheduling using parametric programming. *AIChE Journal*, *54*(10), 2610.

Martínez, E. (1999). Solving batch process scheduling/planning tasks using reinforcement learning. *Computers and Chemical Engineering*, *23*, 527.

Miyashita, K., & Sycara, K. (1994). CABINS: A framework of knowledge acquisition and iterative revision for schedule improvement and iterative repair. *Artificial Intelligence*, *76*, 377.

Miyashita, K. (2000). Learning scheduling control through reinforcements. *International Transactions in Operational Research*, *7*, 125.

Musier, R., & Evans, L. (1989). An approximate method for the production scheduling of industrial batch processes with parallel units. *Computers and Chemical Engineering*, *13*, 229.

Palombarini, J., & Martínez, E. (2010). Learning to repair plans and schedules using a relational (deictic) representation. *Brazilian Journal of Chemical Engineering*, *27*(03), 413.

Pfeiffer, A., Kádár, B., & Monostori, L. (2007). Stability-oriented evaluation of rescheduling strategies, by using simulation. *Computers in Industry*, *58*, 630–643.

Rangsaritratsamee, R., Ferrell, W. G., Jr., & Kurz, M. B. (2004). Dynamic rescheduling that simultaneously considers efficiency and stability. *Computers and Industrial Engineering*, *46*, 1–15.

Shapiro, D., Langley, P., & Shachter, R. (2001). Using background knowledge to speed reinforcement learning in physical agents. In *Fifth international conference on autonomous agents* Montreal, (pp. 254–261).

Sutton, R., & Barto, A. (1998). *Reinforcement learning: An introduction*. MIT Press: Boston.

Trentesaux, D. (2009). Distributed control of production systems. *Engineering Applications of Artificial Intelligence*, *22*, 971.

Van Otterlo, M. (2009). *The logic of adaptive behavior*. IOS Press: Amsterdam.

Vieira, G., Herrmann, J., & Lin, E. (2003). Rescheduling manufacturing systems: A framework of strategies, policies and methods. *Journal of Scheduling*, *6*, 39.

Watkins, C. (1989). *Learning from delayed rewards*. PhD Thesis, Cambridge University.

Wilson, J. (2003). Gantt charts: A centenary appreciation. *European Journal of Operational Research*, *149*, 430–437.

Zaeh, M., Reinhart, G., Ostgathe, M., Geiger, F., & Lau, C. (2010). A holistic approach for the cognitive control of production systems. *Advanced Engineering Informatics*, *24*, 300.

Zhu, G., Bard, J., & Yu, G. (2005). Disruption management for resource-constrained project scheduling. *Journal of the Operational Research Society*, *56*, 365–381.

Zweben, M., Davis, E., Doun, B., & Deale, M. (1993). Iterative repair of scheduling and rescheduling. *IEEE Transactions on Systems, Man and Cybernetics*, *23*, 1588.