Expert Systems with Applications 39 (2012) 10251–10268

Contents lists available at SciVerse ScienceDirect

Expert Systems with Applications

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

SmartGantt – An intelligent system for real time rescheduling based on relational reinforcement learning

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

Keywords:
Manufacturing systems
Real-time rescheduling
Automated planning
Reinforcement learning
Information systems
Relational abstractions

ABSTRACT

With the current trend towards cognitive manufacturing systems to deal with unforeseen events and disturbances that constantly demand real-time repair decisions, learning/reasoning skills and interactive capabilities are important functionalities for rescheduling a shop-floor on the fly taking into account several objectives and goal states. In this work, the automatic generation and update through learning of rescheduling knowledge using simulated transitions of abstract schedule states is proposed. *Deictic* representations of schedules based on focal points are used to define a repair policy which generates a goal-directed sequence of repair operators to face unplanned events and operational disturbances. An industrial example where rescheduling is needed due to the arrival of a new/rush order, or whenever raw material delay/shortage or machine breakdown events occur are discussed using the *SmartGantt* prototype for interactive rescheduling in real-time. *SmartGantt* demonstrates that due date compliance of orders-in-progress, negotiating delivery conditions of new orders and ensuring distributed production control can be dramatically improved by means of relational reinforcement learning and a deictic representation of rescheduling tasks.

1. Introduction

Increasing global competition, a shift from seller markets to buyer markets, mass customization, operational objectives that highlight customer satisfaction and ensuring a highly efficient production, give rise to complex dynamics and on-going 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 supply chains are demanded for products, processes and clients all over the product lifecycle. In this context, established production planning and control systems must cope with unplanned events and intrinsic variability in manufacturing environments where difficult-to-predict circumstances occur as soon as plans are released to the shop-floor (Méndez, Cerdá, Harjunkoski, Grossmann, & Fahl, 2006; Vieira, Herrmann, & Lin, 2003). 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. In this way, for both human planners and shop floor operators (who interpret the plan) uncertainty in a real manufacturing system is a complex phenomenon that cannot be addressed exclusively through the inclusion of uncertain parameters into problem statement (Aytug, Lawley, McKay, Mohan, & Uzsoy, 2005). Several disturbances and events may produce different impacts depending on the context in which they occur, e.g. operators performance may vary during the week or events arising at night may have a greater impact due to the absence of specialized support personnel, as well as the uncertainty that affects materials availability may disrupt production processes in different ways, depending on product recipes.

The vast majority of the scheduling research does not explicitly consider execution issues such as uncertainty, and implicitly assumes that the global schedule will be executed exactly as it emerges from the algorithm that generates it. The existing body of theory does not address different causes, the context in which uncertainty arises, or the various impacts that might result (McKay & Wiers, 2001; Pinedo, 2005, 2008). Moreover, including additional constraints into global scheduling models significantly increases problem complexity and computational burden, of both the schedule generation and rescheduling tasks, which are (in general) NP-hard (Chieh-Sen, Yi-Chen, & Peng-Jen, 2012). Hence, schedules generated under deterministic assumptions are often suboptimal or even infeasible (Henning, 2009; Li & Ierapetritou, 2008; Vieira et al., 2003; Yagmahan & Yenisey, 2010; Zaeh et al., 2010). As a result, reactive scheduling is heavily dependent on the capability of generating and representing knowledge about strategies for repair-based scheduling in real-time. Finally, producing *satisfactory* schedules rather than optimal ones in reasonable computational

* Corresponding author.
E-mail address: ecmarti@santafe-conicet.gov.ar (E. Martínez).

0957-4174/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
doi:10.1016/j.eswa.2012.02.176

time, in an integrated manner with enterprise resource planning and manufacturing execution systems is mandatory for responsiveness (Herroelen & Leus, 2004; Trentesaux, 2009; Vieira et al., 2003).

Reactive scheduling literature mainly aims to exploit peculiarities of the specific problem structure (Adhitya, Srinivasan, & Karimi, 2007; Miyashita, 2000; Miyashita & Sycara, 1995; Zhang & Dietterich, 1995; 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 for generating rescheduling knowledge for specific events. However, the tricky issue is that resorting to a feature-based representation of schedule state is very inefficient, and generalization to unseen schedule states is highly unreliable (Morales, 2004). Therefore, any learning performed and acquired knowledge are difficult to transfer 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. Most of the existing works on rescheduling prioritize schedule efficiency using a mathematical programming approach, in which the repairing logic is not clear to the end-user. In contrast, humans can succeed in rescheduling thousands of tasks and resources by increasingly learning in an interactive way a repair strategy using a natural abstraction of a schedule: a number of objects (tasks and resources) with attributes and relations (precedence, synchronization, etc.) among them. Such conditions, as well as the requirements agility and productivity, together with poor predictability of a shop-floor dynamics, an increasing number of products, reconfigurable manufacturing lines and fluctuations in market conditions demand from production planning and control systems to incorporate higher levels of intelligence.

Today's standard, rigid and hierarchical control architectures in industrial environments have been unable to face with the above challenges, so it is essential to pursue a paradigm shift from off-line planning systems to on-line and closed-loop control systems (Zaeh & Ostgathe, 2009), which take advantage of the ability to act interactively with the user, allowing him to express his preferences in certain points of the decision making process to counteract the effects of unforeseen events, and set different schedule repair goals that prioritize various objectives as such stability, efficiency, or a mix between the two, having into account particular objectives related to customer satisfaction and process efficiency. A promising approach to sustainable improvements in flexibility and adaptability of production systems is the integration of artificial cognitive capabilities, involving perception, reasoning /learning and planning skills (Zaeh et al., 2010). Such ability enables the scheduling system to assess its operation range in an autonomic way, and acquire experience through intensive simulation while performing repair tasks. By integrating learning and planning, the system builds models about the production process, resource and operator capabilities, as well as context information, and at the same time discovers structural patterns and relations using general domain knowledge. Therefore, a scheduling system integrates continuous real-time information from shop-floor sensors/actuators with models that are permanently updated to adapt to a changing environment, and to optimize action selection. At the representation level, it is mandatory to scale up towards a richer language that allows the incorporation of the capabilities mentioned above (Morales, 2003; Van Otterlo, 2009); in that sense, first-order relational representation it's a natural choice because it enables the exploitation of the existence of domain objects and relations (or, properties) over these objects, and make room for *quantification* over objectives (goals), action effects and properties of schedule states (Blockeel, De Raedt, Jacobs, & Demoen, 1999; Džeroski, De Raedt, & Driessens, 2001).

In this work, a novel real-time rescheduling prototype application called *SmartGantt*, which resorts to a relational (deictic) representation of (abstract) schedule states and repair operators with RRL is presented. To learn a near-optimal policy for rescheduling using simulations (Croonenborghs, 2009), an interactive repair-based strategy bearing in mind different goals and scenarios is proposed. To this aim, domain-specific knowledge for reactive scheduling is developed using two general-purpose algorithms already available: TILDE and TG (De Raedt, 2008; Džeroski et al., 2001).

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

Fig. 1 depicts the repair-based architecture implemented by *SmartGantt*, embedded in a more general setting including an Enterprise Resource Planning System (ERP) and a Manufacturing Execution System with communication and control infrastructures, which integrates artificial cognitive capabilities in resources and processes to include by design flexibility and adaptability in production systems (Trentesaux, 2009). In this approach, search control knowledge about optimal selection of repair operators is generated through reinforcements using a schedule state simulator. 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 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 applied is performed by *SmartGantt* 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. Such an update is made using a reinforcement learning algorithm (Sutton & Barto, 1998) such as the well-known Q-learning rule, which is showed in Fig. 2. By accumulating enough experiences over many simulated transitions, *SmartGantt* is able to learn an optimal policy for choosing the best repair operator at each schedule state.

The main benefit of applying reinforcement learning techniques such as Q-learning to search control knowledge for improving quality and efficiency of real-time rescheduling is that there is no extra burden on the availability of domain experts, allows online adaptation to a dynamic environment and make room for abstractions that are necessary to deal with large state spaces, e.g. supply chains. For repairing a schedule, *SmartGantt* is given a repair-based *goal* function:

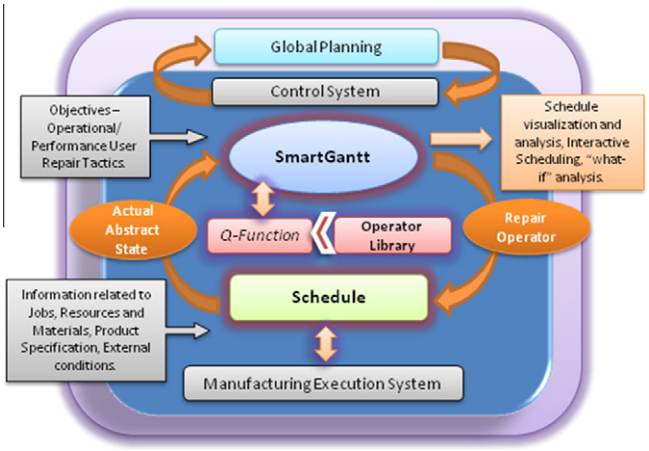


Fig. 1. Repair-based architecture implemented by *SmartGantt*.

```
for each s ∈S and a∈A do
      initialize table entre Q(s, a)
end for
generate a starting state s
repeat
      select an action a and execute it
      receive an inmediate reward r=r(s, a)
      observe the new state s'
      update the table entry for Q(s, a) as follows:
      Q(s,a) ← r + γ max_b Q(s',b)
      s← s'
until no more learning episodes
```

Fig. 2. Basic Q-learning algorithm.

$$goal : S \rightarrow \{true, false\} \quad (1)$$

defining which states in the repaired schedule are target states, e.g. states where total tardiness is less than or equal to 1 working day. Usually, a precondition function like (2):

$$pre : S \times A \rightarrow \{true, false\} \quad (2)$$

is used to specify which subset of repair operators can be applied at each state of the schedule to account for resource capabilities and precedence constraints (e.g. product recipes) as well as alternative repair actions such split a batch, merge batches, shifting a task between resources, etc. 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:

$$goal(\delta(\ldots\delta(s_1, a_1)\ldots, a_n)) = true \quad (3)$$

where δ is the transition function which is only revealed to the learning agent through simulation.

The reward function is used to translate the goal in Eq. (3) into a reinforcement signal to guide the search for a policy that select the best repair operator in each schedule state (Martinez, 1999). Resorting to the reward function and simulations, the optimal *policy*: $a_i = \pi s_i$ is approximated here using a regression tree (Blockeel & De Raedt, 1998; Van Otterlo, 2009), and then used to compute the action-sequence to reach a repaired scheduled, improving responsiveness at the shop-floor to handle unplanned events and rejecting disturbances at the shop-floor (Palombarini & Martínez, 2010). In turn, different operational and performance objectives, as well as user preferences and customized repair tactics can be provided by a human expert to *SmartGantt*. The latter provides visualization, alternative solutions and *what–if* analysis capabilities, so that the user can work in a fully interactive fashion by using a graphical interface that allows achieving alternative rescheduling goals.

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), and can be defined formally as follows (Van Otterlo, 2009):

Definition 1. Let $P = \{p_1/\alpha_1, \ldots, p_n/\alpha_n\}$ be a set of first order predicates with their arities, $C = \{c_1, \ldots, c_k\}$ a set of constants, and let $A' = \{a_1/\alpha_1, \ldots, a_m/\alpha_m\}$ be a set of actions with their arities. Let S' be the set of all ground atoms that can be constructed from P and C, and let A be the set of all ground atoms over A' and C. A Relational Markov Decision Process (RMDP) is a tuple $M = \langle S, A, T, R\rangle$, where S is a subset of S', A is defined as stated, $T: S \times A \times S \rightarrow [0,1]$ is a probabilistic transition function and $R: S \times A \times S \rightarrow IR$ a reward function.

The difference between RMDPs and MDPs is the definition of S and A, whereas T and R are defined as usual. Formulating the reschedul-

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. 3. A RRL algorithm for learning to repair schedules through intensive simulations.

ing problem as a RMDP enables *SmartGantt* to rely upon relational abstractions of the state and action spaces to reduce the size of the learning problem. RMDP offers many possibilities for generalization due to the structured form of ground atoms in the states and actions spaces, which share parts of the problem structure (e.g. constants). 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 below.

Example 1. **state1** = `focal(task(task1,product(a))),resource(1, extruder(1), [task(task1, product(a)), task(task2, product(b)), task(task5, product(a))]), resource(2, extruder(2), [task(task3, product(a)), task(task4, product(c)), task(task7, product(c))])`; **action1** = `action(rightMove(task(task1), task(task2))))`.

Hence, RRL algorithms are concerned with reinforcement learning in domains that exhibit structural properties and in which different kinds of related objects, namely tasks and resources exist (De Raedt, 2008; Džeroski et al., 2001; 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. In this kind of environments, most traditional reinforcement learning techniques break down, because they generally store the learned *Q*-values explicitly in a state-action table, with one value for each possible combination of states and actions. Rather than using an explicit state–action *Q*-table, RRL stores the *Q*-values in a logical regression tree (Blockeel & De Raedt, 1998). The relational version of the *Q*-learning algorithm is shown in Fig. 3.

The computational implementation of the RRL algorithm has to deal successfully with the relational format for (*states*, *actions*)-pairs in which the examples are represented and the fact that the learner is given a continuous stream of (*state*, *action*, *q-value*)-triplets to learn predicting *q-values* for (*state*, *action*)-pairs during training.

Because of the relational representation of states and actions and the inductive logic programming component of the RRL algorithm, there must exist some body of *background knowledge* which is generally true for the entire domain to facilitate induction. After the *Q*-function hypothesis has been initialized, the RRL algorithm starts running learning episodes (Džeroski et al., 2001; Sutton & Barto, 1998). During each episode, all the encountered states and the selected actions are stored, together with the rewards related to each visited (*state*, *action*)-pair. At the end of each episode, when the system encounters a *goal* state, it uses reward back-propagation and the current *Q*-function approximation to compute and update the corresponding *Q*-value approximation for each encountered (*state*, *action*)-pair in the episode The algorithm presents the set of (*state*, *action*, *q-value*)-triplets encountered in each learning episode to a relational regression engine, which will use this set of *Examples* to update the current regression tree of the *Q*-function.

TG relational regression algorithm (De Raedt, 2008; Driessens, Ramon, & Blockeel, 2001; Sutton & Barto, 1998) is used by *SmartGantt* for accumulating simulated experience in a compact way, yet readily available decision-making rule for generating a sequence of repair operators available at each schedule state *s*. Such experience is stored in a first-order decision tree (FODT), in which every internal node contains a test which is a conjunction of first-order literals (see Fig. 4). Also, every leaf (terminal node) of the tree involves a prediction (nominal for classification trees and real valued for regression trees). Prediction with first-order trees is similar to prediction with propositional decision trees: every new instance is sorted down the tree. If the conjunction in a given node succeeds (fails) for that instance, it is propagated to the left (right) subtree. This FODT is converted by *SmartGantt* in a set of Prolog rules that is used in execution time to predict *Q*-values and select repair operators accordingly.

The TG algorithm starts with the tree containing a single node, all examples and the Root Query, and then recursively completes the rest of nodes. It is important to define correctly the Root Query, because it is the first node of the tree, and the basis upon which further refinements will be performed. Furthermore, the Root Query generates a set of basic variables about which TG may execute several tests to build the rest of the regression tree. As a consequence, in the derived set of Prolog rules, the Root Query is present in the first part of the antecedent of each one of them. In this work, the query defined as a root is showed in the Example 2:

Example 2. `root((focal_Task(T),totalTardiness(W)maxTardiness (X),avgTardiness(Y),totalWorkInProcess(Z),tardinessRatio(A), inventoryRatio(B), totalCleanoutTime(F), focalTardiness(G), action_move(Cons,T,SF))).`

In order to complete a node, the algorithm first tests whether the example set in the node is sufficiently homogeneous. If it is, the node is turned into a leaf; if it is not, all possible tests for the node are computed and scored using a heuristic. This possible test are taken from the background knowledge definition, which is explained below, and can be relational facts, queries about the value of a discretized variable, or more complex predicates that can involve several rules. Then, the best test is added into the node, and two new nodes are incorporated to the tree: the left one contains those examples for which the test has been successful and the right one those for which the test fails. The procedure is then called recursively for the two sub-nodes. Once the instance arrives in a leaf node, the value of that leaf is used as the prediction for that instance. The main difference between this algorithm and traditional decision tree learners relies in the generation of the tests to be incorporated into the nodes. To this aim, the algorithm employs a refinement operator ρ that works under θ-*subsumption*. Therefore, the refinement operator specializes a query *Query* (a set of literals) by adding literals *lits* to the query yielding *Query, lits*. An example for the query ← precedes(*X*,*Y*) is showed in Table 1.

Related to the use of the operators, the procedure needs to be able to propagate the query along the succeeding branches in the tree. This propagation allows the binding of the variables between

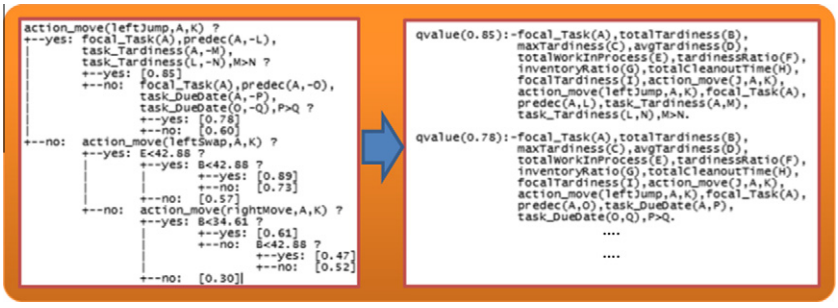


Fig. 4. A simple example of a relational regression tree (left), and a detail of two from ten possible derived Prolog rules (right).

Table 1
Example of several refinement operations for the query precedes(*X*, *Y*).

Query	Refinement
← precedes(X,Y)	← precedes(*X*, *Y*), orderOfProduct(*X*, Product) ← precedes(*X*, *Y*), task_in_resource(*X*, Resource) ← precedes(*X*, *Y*), task_DueDate(*X*, DueDate) ← precedes(*X*, *Y*), task_Time(*X*, Time) ← precedes(*X*, *Y*), task_DueDate(*X*, DueDate)

the different tests. Several heuristic functions can be used to determine the best tests, and to decide when to turn nodes into leaves. The function employed by TG is based on information gain, which measures the amount of information gained by performing a particular test. The entropy (or information) $I(P,N)$ needed to classify an example in one of two classes P and N (with $E = P \cup N$) is defined in Eq. (4) (De Raedt, 2008)

$$I(P,N) = -\frac{n(P)}{n(E)} \times \log_2 \frac{n(P)}{n(E)} - \frac{n(N)}{n(E)} \times \log_2 \frac{n(N)}{n(E)} \quad (4)$$

where P and N are the sets of positive and negative examples, respectively, and $n(X)$ denotes the number of examples in the set X. Furthermore, if the set of examples $E = P \cup N$ is splitted into the sets $El = P_l \cup N_l$ and $E_r = P_r \cup N_r$ with regard to the test t then the information gain can be expressed as is showed in Eq. (5) (De Raedt, 2008).

$$IG(E,E_l,E_r) = I(P,N) - \frac{n(E_l)}{n(E)} \times I(P_l,N_l) - \frac{n(E_r)}{n(E)} \times I(P_r,N_r) \quad (5)$$

In the above Eq. (5), IG measures how much information is gained by performing the test t. Thus, a decision tree learning algorithm selects the test resulting in the maximal information gain. So, TG stores the current tree together with statistics for all tests that can be used to decide how to split each leaf further. Every time an example (triplet) is inserted, it is sorted down the tree according to the tests in the internal nodes and, in the resulting leaf the statistics of the tests are updated.

3.1. Relational (deictic) representation of schedule states and repair operators

The drawbacks of attribute-value representations in learning a rescheduling policy that have been described in previous sections are solved by *SmartGantt* by resorting to *relational* (or *first-order*) deictic representations. This approach, relies to a language for expressing sets of relational facts that describe schedule states and actions in a compact and logical way; each state is characterized by only those facts that hold in it, which are obtained applying a *hold(State)* function. Formally, *first-order* representations are based in a relational alphabet Σ, which consists of a set of relation symbols $\mathtt{P}$ and a set of constants $\mathtt{C}$. Each constant $\mathtt{c} \in \mathtt{C}$ denotes an object (i.e. a task or resource) in the domain and each $\mathtt{p/a} \in \mathtt{P}$ denotes either a property (or attribute, i.e. task tardiness) of some object (if $\mathtt{a} = 1$) or a relation between objects (for example, if $\mathtt{a} > 1$, e.g. $\mathtt{precedes(taskl\ ,task2)}$).

To represent *structured terms* in the schedule domain, e.g. resource $(\mathtt{l}, \mathtt{extruder}(1), [\mathtt{taskl}, \mathtt{task2}, \mathtt{task5}])$, the relational alphabet is extended with a set of *function symbols or functors* $F = \{f_1/\alpha_1, \ldots, f_k/\alpha_k\}$ where each $f_i (i = 1, \ldots, k)$ denotes a function from C_k to C, where α is called the "arity" of the functor, and fixes the number of its arguments, e.g. precedes/2, task/5, averageTardiness/1, focalRightSwappability/0, among others. *SmartGantt* implements the concept of "learning from interpretations" (Blockeel et al., 1999; De Raedt & Džeroski, 1994), so in this notation, each (state, action) pair will be represented as a set of relational facts, which is called a *relational interpretation*.

In addition to a relational abstraction, a deictic representation for describing schedule states and repair operators is proposed as a powerful alternative used by *SmartGantt* to scale up RRL in rescheduling problems. Deictic representations deal naturally with the varying number of tasks and resources in the planning world by defining a focal point for referencing objects (tasks and resources) in the schedule. This focal point is represented by a functor called focal/1, which takes one parameter to specify a task that fixes the repair scope and objectives. Such task is selected by *SmartGantt* using different criteria, depending on the type of event which generates the disruption. Once this focal task is known, other facts that describe the schedule state and are relevant for repairing it can be established, such as leftTardiness/1 or altRightTardiness/1, among others. So, to characterize transitions in the schedule state due to repair actions, a deictic representation resorts to constructs such as: (i) The first task in the new order, (ii) The next task to be processed in the affected resource, and (iii) Tasks related to the last order in the priority order.

Fig. 5 provides a relational (deictic) representation of a schedule. Note that the number of facts in an example is not fixed, and the order of them is arbitrary; then each state can have a varying number and type of relationships that characterize it, providing a higher level of flexibility and allowing *SmartGantt* to deal with situations where there exist uncertainty and incomplete information. Furthermore, the relational nature of states and actions gives rise to structural patterns that can be exploited by *SmartGantt* to define compact abstractions, inducing Prolog logical rules, something that is not possible in a propositional representation of rescheduling problems. For example, the structure $\mathtt{precedes(task(taskl), task(task2))}$ shares the parameter object "task(task2)" with the fact $\mathtt{precedes(task(task2), task(task3))}$; it can be generalized

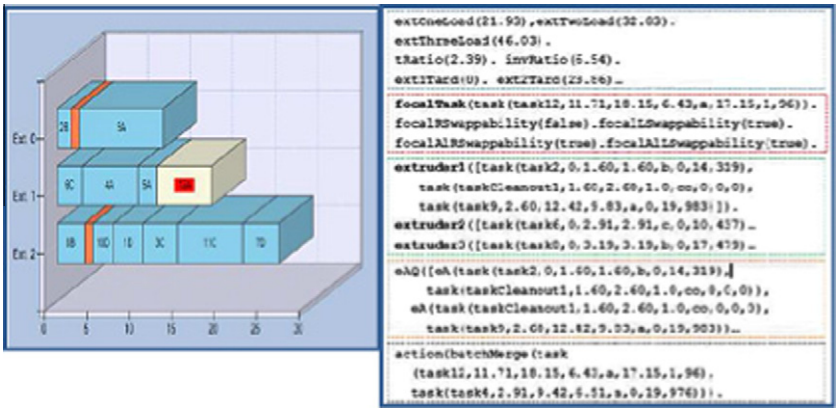


Fig. 5. Relational representation of a schedule state.

by the logical conjunction $\texttt{precedes}(A,B),\texttt{precedes}(B,C)$ where *A*, *B* and *C* are unbounded variables which can represent any object of type "task", covering several examples of different facts where this relation aptly applies.

Summing up, in a deictic representation, both scheduling states and repair operators (actions) are defined in relation to a given focal point (i.e. a task, group of tasks, a resource, or a virtual production line) as it is shown in Fig. 6. 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 may need to be moved as well which is 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.

3.2. Induction of abstract schedule states and repair policy using logical decision trees

Relational representations of schedule states and repair operators are *symbolic* in nature; therefore, many methods have been developed to perform the process of generalizing and abstracting over them, like Distances and Gaussian Kernels for structured data (Gärtner, 2008) and Relational Instance Based (RIB) regression (Driessens, 2004). In this work, this task is carried on by *SmartGantt* using the result of applying TG, in combination with the algorithm depicted in Fig. 7, based mainly in the concept of first-order abstractions of version spaces (Mitchell, 1997).

In simulation-based generation of rescheduling knowledge the main challenge is to exploit the inherent structure of relational schedule states, as well as the structure shared among several schedules or parts of them (i.e. precedence *o* parallelism relations between tasks in production recipes) using *variables* to range over constants, and background knowledge with syntactic bias in the form of types and modes (i.e. declarations written as $type(pred(type_1,\ldots,type_n))$, where *pred* denotes the name of the predicate and the $type_i$ denote the names of the types), that consists of the definitions of general predicates that can be used in the induced hypotheses, for generalization, abstraction, and knowledge transferring purposes (Croonenborghs, Driessens, & Bruynooghe, 2008; Morales, 2004; Torrey, Shavlik, Walker, & Maclin, 2006).

As can be seen in Fig. 8, each abstract state models a set of interpretations of the underlying learning process (RMDP), and defines which relations should hold in each of the states it covers. For

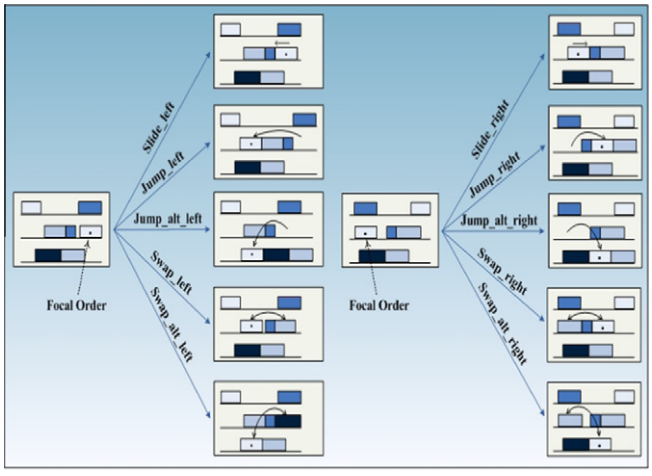


Fig. 6. Deictic repair operators.

```
Initialize
s ← actual schedule ground state in relational format.
P ← collection of prolog rules induced by TG, that
represents the repair policy.
BK ← collection of background knowledge rules
ABS ← empty collection

For each rule available in P
      AbstractState ← body(rule)
      ABS.Add(AbstractState)
Next
Consult BK, ABS
For each abstractstate available in ABS
      if abstractstate θ-subsumes s then
            return abstractstate
      end if
Next
return ∅
```

Fig. 7. Algorithm that determines the abstract state corresponding to a ground state.

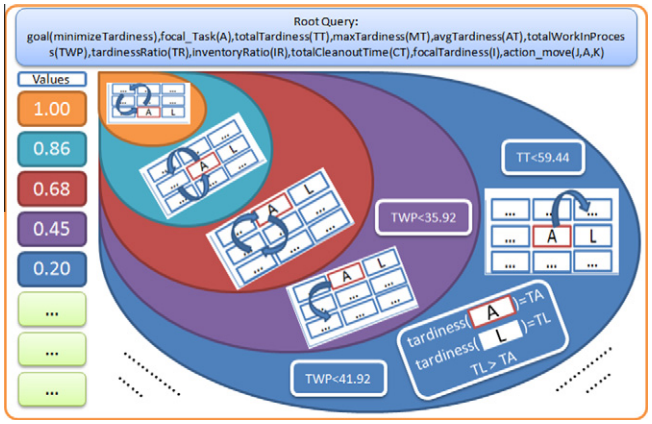


Fig. 8. Part of the abstract state action-value function for the minimize tardiness goal.

example, in Fig. 8, the blue region covers the set of states where Total Tardiness is less than 59.44 h, the focal task *A* precedes a certain task *L*, and has less tardiness than task *L*, the total Work in Process is less than 41.92 h, and the repair operator that has been applied is *JumpAltRight*. Formally, this is expressed as a conjunction $\equiv 1\wedge\cdots\wedge m$ of logical atoms, e.g. a logical query. The use of variables, as is showed in the same figure in the Root Query, admits abstracting over specific domain objects as well. The process starts with a first-order logical alphabet containing predicates, constants and variables. Thus, an abstract state is basically a logical sentence, specifying general properties of several states visited during learning through simulation of abstract state transitions. An example is showed in Example 3.

Example 3. $\forall A,B,C,D(\texttt{avgTard}(A),\texttt{precedes}(B,C),\texttt{precedes}(C,D),A>57.8)$ is an Abstract State that denotes the set of schedules states in which the average tardiness *A* is greater than 57.8, where a task *B* precedes a task *C*, who in turns precedes the task *D*. Note that in this case, the task denoted by *C* must be the same in all cases where it appears.

Fig. 8 highlights that the action-value *Q*-function relies on a set of abstract states, which together encode the kind of rescheduling knowledge learned through intensive simulation in a compact and formal way, which can be used in real time to repair plans whose feasible have been affected by disruptive events. Furthermore, the definition of an abstract state must be 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, it is no mandatory to identify the event type that has driven the schedule to the current state in order to find a sequence of repair operators to achieve a goal for the repaired schedule. However it is relevant to know the available resource (options) to undertake the repair task realistically. In other sense, the abstract state S represents *partial knowledge* about an actual schedule state, and at the same time aggregates a set of state atoms into an equivalence class $[S]$. Using this powerful abstraction, schedule states are characterized by a set of common properties and the obtained repair policy expresses its generalization capability based on the problem structure and relations among objects in the schedule domain (Morales, 2004), which makes room for transferring the rescheduling policy to somewhat different problems were the same relations apply, and without any further learning. As only fully observable worlds are considered, this partial knowledge is only due to abstracting schedules using relationships between concerned objects. An abstract state S covers a ground state s iff $s| = S$, which is decided by *SmartGantt* using θ-subsumption. An example of an induced abstract state is shown in Fig. 9.

Abstract state spaces compactly specify in a logical way a Relational Markov Decision Process state space S as a set of abstract states, and can be defined formally as follows (Van Otterlo, 2009):

Definition 2. Let Γ be a logical vocabulary and let $\mathbb{A}$ be the set of all Γ-structures $\mathcal{A}$, a **multi-part abstraction** (MPA) over $\mathbb{A}$ is a list $[\varphi_1, \ldots, \varphi_n]$, where each $\varphi_i (i = 1 \ldots n)$ (called a **part**) is a formula. A structure $\mathcal{A} \in \mathbb{A}$ is covered by an MPA iff there exists a part $\varphi_i (i = 1 \ldots n)$ such that $\mathcal{A}| = \varphi_i$. An MPA is a partition iff for all structures there is exactly one part that covers it. An MPA μ over Σ induces a set of equivalence classes over Σ, possibly forming a partition. MPAs are to be seen as sets. In other words, μ is a compact representation of a first-order abstraction level over Σ. An element $\sigma \in \Sigma$ is covered by a part $< \varphi >$ iff σ |=φ.

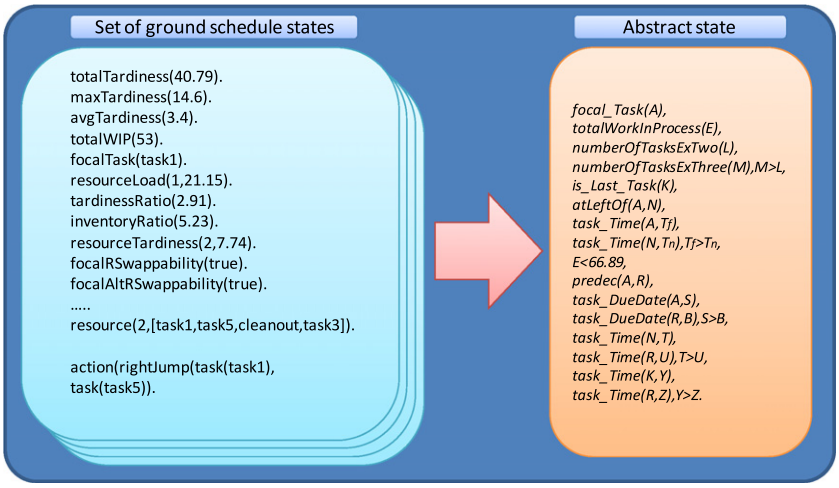


Fig. 9. Induction of several ground states in an abstract state.

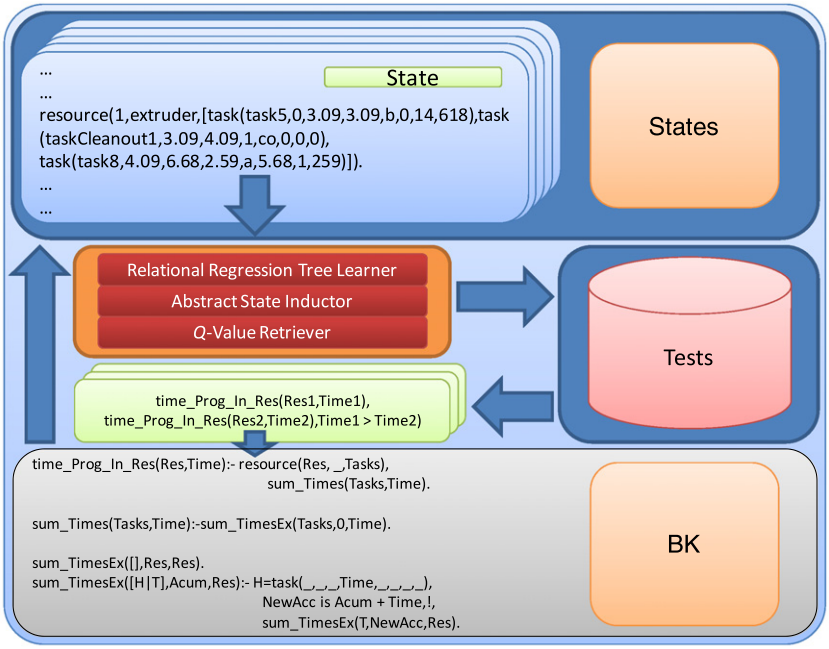


Fig. 10. Test checking using background knowledge.

Table 2
Some examples of the tests designed.

Definition	Interpretation
rmode(#(1 * 10 * C: threshold(totalTardiness(W), [W], C), +W < C))	Get values of the discretized variable totalTardiness
rmode(10: (focal_Task(+T), predec(T, −T2), task_Tardiness(T, −Number), task_Tardiness(T2, −Number1), Number > Number1))	Focal Order T is predecessor of T2 and has more tardiness
rmode(10: (focal_Task(+T), atLeftOf(T, −T2), task_Tardiness(T, −Number), task_Tardiness(T2, −Number1), Number > Number1))	Focal Order T is at left of T2 and its tardiness is greater than the last
rmode(10: (order_of_product(+O,+A)))	Order O is of product A
rmode(10: (task_in_resource(+T,+R)))	Task T is scheduled to be processed in the resource R
rmode(10: (task_Time(+T, −N), task_Time(+T1, −N1), N > N1))	Processing time of task T is greater than the processing time of task T1
rmode(10: (task_Tardiness(+T, −N), task_Tardiness(+T1, −N1), N > N1))	Tardiness of task T is greater than the tardiness of task T1
rmode(10: (task_DueDate(+T, −N), task_DueDate(+T1, −N1), N > N1))	Due date of task T is greater than the due date of task T1
rmode(10: (focal_Task(+T), task(−T1), task_Time(T, −Time1), task_Time(T1, −Time2), Time1 > Time2))	Processing time of focal task T is greater than the processing time of task T1
rmode(10: (time_on_resource(−R, −Time), time_on_resource(−R1, −Time1), Time < Time1))	Total processing time of tasks scheduled in resource R is minor than the Total processing time of tasks scheduled in resource R1
rmode(10: (tard_on_resource(−R, −Tard), tard_on_resource(−R1, −Tard1), Tard < Tard1))	Total tardiness of tasks scheduled in resource R is minor than the Total tardiness time of tasks scheduled in resource R1
rmode(10: (time_Prog_In_Res(Res1, Time1), time_Prog_In_Res(Res2, Time2), Time1 > Time2))	The total time of the scheduled tasks in Res1 is greater than the programmed into Res2

Based on the presented RRL approach, *SmarttGantt* generates the definition of the *Q*-function from a set of examples in the form of abstract state-action-value tuples, and dynamically makes partitions of the set of possible states, as can be seen in Fig. 10. These partitions are described by a kind of *abstract* schedule state, that is, a logical condition, which matches several real schedule states. For instance, the condition that is shown at right of Fig. 9, matches all states in which *A* is the focal task, *E* is the total WIP, *L* is the number of tasks that are in the queue for resource #2, and *M* is the corresponding number of tasks in resource #3, *M* is greater than *L*, the focal task is planned before than task *N*, *K* is the last task that will be carried out in the corresponding resource, the process time for the focal task is t_f, the process time for the task *N* is t_N, t_f is greater than t_N, the total WIP is less than 66.89, the focal task precedes a task *R*, the due date of the focal task is greater than the due date for *R*, the process time of task *N* is greater than the process time for *R*, and the process time for *K* is greater than the process time for *R*. The relational *Q*-learning approach sketched above thus needs to solve two tasks: finding the right partition and learning the right values for the corresponding abstract state-action pairs. The abstract *Q*-learning algorithm starts from a partition of the state space in the form of a decision list of abstract state-action pairs $((S_1, A_1), \ldots, (S_n, A_n))$ where is assumed that all possible abstract actions A_i are listed for all abstract states S_i. Each abstract state S_i is a conjunctive query, and each abstract action A_i contains a possibly *variabilized* action. The relational *Q*-learning algorithm now turns the decision list into the definition of the qvalue/1 predicate, and then applies *Q*-learning using the qvalue/1 predicate to rank state-action pairs. This means that every time a concrete state-action pair (s, a) is encountered, a *Q*-value *q* is computed using the current definition of qvalue/1, and then the abstract *Q*-function, that is, the definition of qvalue/1 is updated for the abstract state-action pair to which (s, a) belongs.

3.3. Background knowledge

As it was previously mentioned, 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 that represent *Q*-functions, and to evaluate if a ground 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 in the abstract states and repair actions. This is a powerful feature of inductive logic programming approach used in this here to keep simple test definitions which to be useful has to be previously derived in terms of other relationships that must be true in the state that is been evaluated. Fig. 10 highlights how the test **T1**:

test(time_Prog_In_Res(Res1,Time1),time_Prog_In_Res(Res2,Time2), Time1 > Time2)

is 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 doesn't exist on the set of facts that determine the schedule state, but is based on a Prolog rule defined in the BK. This rule needs to obtain the list of tasks programmed in a resource - using resource(Res,_,Tasks), which exists in the state definition- and thereafter calculate 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 *Q*-Value retrieving). Furthermore, *BK* also might specify constraints such as

$$\textbf{BK}: \forall\, XY(\texttt{precedes}(X, Y) \rightarrow \texttt{X} / = \texttt{Y}),$$

saying that if task `X` precedes task `Y`, they should be different objects. In Table 2, some examples of the designed BK are showed.

In consequence, BK can cover different types of information for a given domain, like experts or users having general domain knowledge, value function information, partial knowledge about the application of a repair operator, similarities between different states, or new relations derived of the relational states. In Table 1, some examples of the BK designed are given. The main advan-

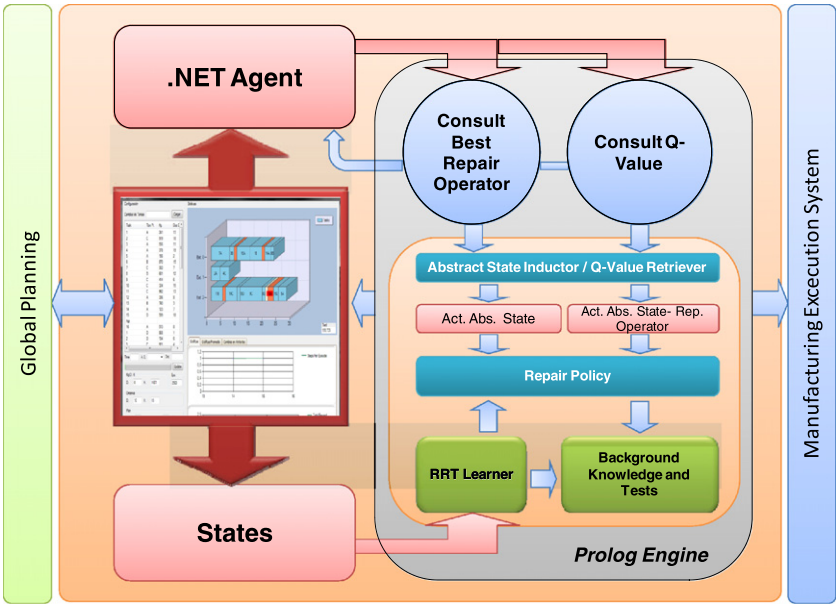


Fig. 11. Schema of the *SmartGantt* architecture.

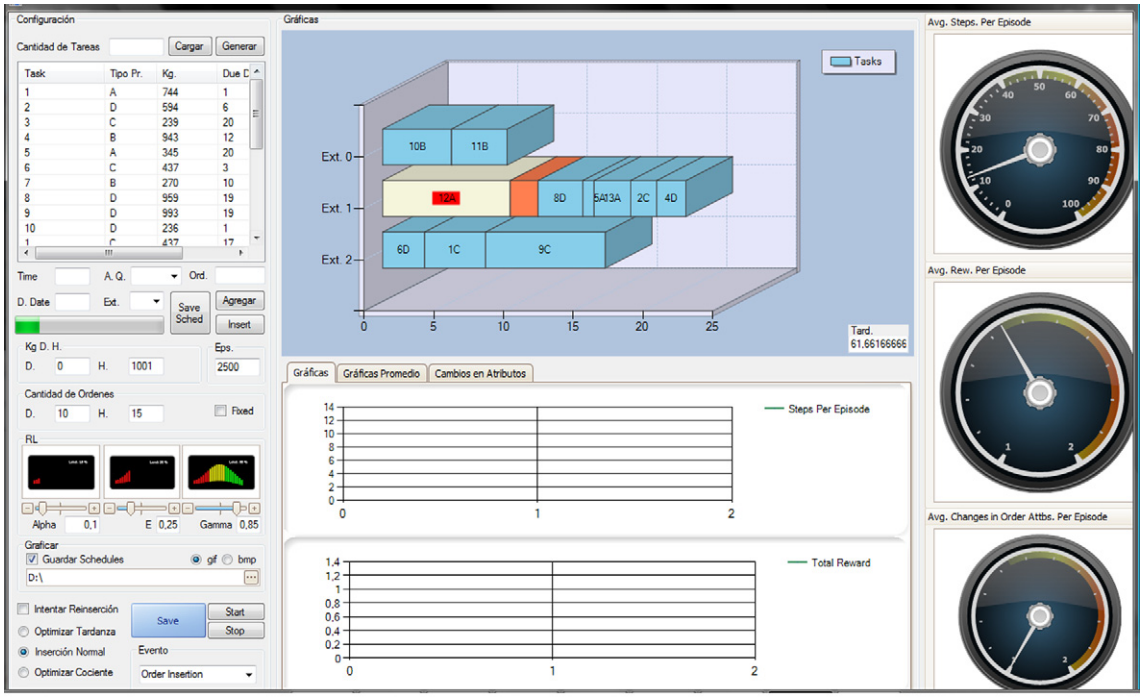


Fig. 12. Graphical user interface.

tage of resorting to BK in *SmartGantt* is that different relationships can be inferred between domain objects (tasks and resources) which affect decisively in the 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. Then, 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 restrict the types of queries that can be generated. In Table 2, the types and modes are included in the definition of predicates.

4. *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. Also, the **TILDE** and **TG** modules from The **ACE Datamining System** developed by the Machine Learning group at the University of Leuven have been used. The overall architecture of the prototype is shown in Fig. 11 whereas

Fig. 12 depicts the graphical user interface. The prototype admits two modes of use: *training* and *consult*. During training *SmartGantt* learns to repair schedules through simulated transitions of schedule states, and the generated knowledge is encoded in the Q-function. Exploitation of rescheduling knowledge is made in the consult interaction. The disruptive events that the system can handle are the arrival of a new order/rush order to the production system, delay or shortage in the arrival of raw materials, and machine breakdown.

Before starting a training session the user must define through the graphical interface, the value of all simulation and training parameters, related to:

- *Initial schedule conditions*: for simulation purposes, is necessary to determine the minimum and maximum values associated to the size and due date that the automatically generated orders can adopt, as well as the (variable) number of orders that may be present in the system at the beginning of the training episode.
- *Learning parameters*: by means of graphical sliders, the user must adjust the RRL associated parameters, like γ, ε, and α.
- *Training results folder*: the user can select the folder where the system stores the images that it generates as a result of training, and its format (*gif* or *bmp*).
- *Goal state definition*: this key parameter that has to be established before starting simulation since it establishes the desired repair goal. Checking the option "Try Reinsertion," enables the system to change the due date of the new order to define the minimum value that this attribute should take so the order can be inserted in the actual schedule conditions. This feature is very import for due date negotiation purposes whenever the order cannot be inserted with its initial requirements.

In the current version of *SmartGantt*, training a rescheduling agent can be carried out by selecting one of three alternative goals, which is selected through an option list:

- *Tardiness improvement*: the repaired schedule should have less tardiness than the initial one. This goal prioritizes the efficiency of the schedule.
- *Stability*: tries to minimize the number of changes made to the initial schedule, and rewards are assigned according to (6), 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 original schedule. The stability goal tries to minimize the impact of changes to the original schedule:

$$r = \begin{cases} \frac{n_0 - n_c}{n_0} & \text{if goal}(s_t) = \text{true} \\ 0 & \text{if goal}(s_t) = \text{false} \end{cases} \quad (6)$$

- *Balancing*: tries to trade off tardiness with the number of changes made to the original schedule (stability goal) in order to achieve the goal. Thus, rewards are defined as follows:

$$r = \frac{T_{\text{Initial}} - T_{\text{Final}}}{N} \quad (7)$$

where N is the number of required steps for achieving the goal state. In all cases, each option means to change the conditions that define the goal state and the way in which rewards are assigned to applied repair operators, depending on the obtained results. For example, in the case of Tardiness Reduction, credit assignment has the particularity of penalizing sequences of repair actions leading to a final state where the total tardiness is greater than the one for initial schedule.

At the beginning of each training episode, the prototype generates a set of tasks for arriving orders with their corresponding attributes, bounded within their allowable ranges defined interactively. To generate the initial schedule state s_0, these tasks are randomly assigned to one of the available resources. Later on, if the selected disruptive event is the arrival of a new order, or rush order, the attributes of the order to be inserted – without increasing the total tardiness in the initial schedule – are generated, and it is assigned arbitrarily to one of the available resources (extruders). If the disruptive event is machine breakdown, the system randomly selects one resource, and generates the breakdown time. Then, the focal task is selected among the available ones. The remaining case occurs when the disruptive event is shortage or delay in the arrival of raw materials; in this situation, the system select randomly a task, and set a delay; then, the selected task is taken as focal point for rescheduling. The learning episode progresses by applying a sequence of repair operators until the goal state is reached. Despite the goal is selected interactively – depending on the disruptive event – there may be stringent situations in which the system cannot simultaneously meet the requirements stated by the user and the goal state, e.g. when the selected goal is Tardiness Improvement, and the disruptive event is machine breakdown. In these cases *SmartGantt* automatically relaxes the goal proposed by the user, and flags a message to make the user aware of the change. Fig 13 shows schematically the elements related to each phase in the interaction between an user and the leaning system.

Based on the suitable initialization of the Q-function, the RRL algorithm starts to simulate learning episodes, while updating its knowledge base using the standard Q-learning algorithm. In each training episode, every found state-action pair is stored with the associated reward. At the end of the episode, when the agent has reached the goal state, the Q-values of each state-action pair visited are updated using back-propagation. Then, the algorithm feeds the set of tuples (state, action, Q) to a relational regression engine (RRTL component in the architecture) in order to update the regression tree that represents the Q-function. Then, *SmartGantt* continues executing the next episode. In the obtained tree, different nodes are basically Prolog queries. In consequence, to find a Q-value, a Prolog program based on a tree which is built on-line (shown in Fig. 13 as Policy(Prolog Rules)). After that, when the query (state-action) is executed, this engine returns the desired value, or the best repair operator for a given schedule state. The estimated Q-value will depend on the quality of the generated tree, which rely upon defined rules in the background knowledge, which is valid all over the domain. Fig. 14 shows an example of an induced rule, although the policy can contain hundreds of them.

The logical in Fig. 14 sequence must be interpreted as follows: "If the Focal Task is *A*, Total Tardiness is *B*, Maximum Tardiness is *C*, Average Tardiness is *D*, WIP is *E*, Tardiness Ratio is *F*, Inventory

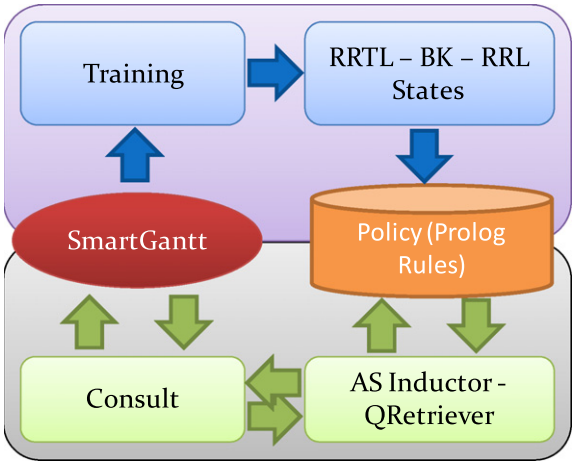


Fig. 13. *SmartGantt* consult and training phases.

```
qvalue(0.84):-
        focal_Task(A),totalTardiness(B),maxTardiness(C),avgTardiness(D),
        totalWorkInProcess(E),tardinessRatio(F),inventoryRatio(G),
        totalCleanoutTime(H),focalTardiness(I),action_move(J,A,K),
        numberOfTasks(Res1,L),numberOfTasks(Res2,M),M>L,
        is_Last_Task(K),atLeftOf(A,N),task_Time(A,O),task_Time(N,P),
        O>P,E<66.89,focal_Task(A),predec(A,Q),task_DueDate(A,R),
        task_DueDate(Q,S),R>S,task_Time(N,T),task_Time(Q,U),T>U,
        E<61.72,focal_Task(A),atLeftOf(A,V),task_DueDate(A,W),
        task_DueDate(V,X),W<X,task_Time(K,Y),task_Time(Q,Z),Y>Z,
        action_move(rightJump,A,K), !.
```

Fig. 14. An induced Prolog rule.

Ratio is *G*, Total CleanOutTime is *H*, Tardiness of Focal Task is *I*, the repair operator involves the tasks *A* and *K*, the number of tasks present in resource Res1 is greater than the number of tasks present in resource Res2, Task *K* is in the last position of the resource, Focal Task *A* begins before of a Task *N* which has a lower processing time, $E < 66.89$, Focal Task *A* precedes a Task *Q* whose due date is less, Task *N* has a processing time greater than *Q*, $E < 61.72$, Focal Task *A* begins before than a Task *V* and has a lower due date, processing time of Task *K* is greater than processing time of Task *Q*, if the repair operator is RightJump, then *Q* Value is 0.84".

The relational regression tree contains (in relational format) the repair policy learned from the training episodes. The mentioned queries are actually processed by the Prolog wrappers **QManager.dll** and **OperatorManager.dll,** which made up a transparent interface between the .NET agent and the relational repair policy and objects describing schedule 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. The algorithm depicted in Fig. 7 is implemented in a separated dynamic library that uses the functionality of Prolog.NET to perform the induction of abstract states.

Fig. 15 shows a component diagram for *SmartGantt*. In the .NET prototype, different classes are used to model **Agent**, **Environment**, **Actions** and **Policy** concepts. Furthermore, *SmartGantt* is able to access relational objects using a parser designed specifically to transform the decision tree in Prolog rules. This parser is embedded in the QManager component. The remaining components use the files Policy.pl, ActState.pl, ActStateAction.pl and BackgroundKnowledge.pl. Some of them are modified dynamically in execution time. Finally, the .NET agent is fully equipped to handle situations where the rescheduling goal is not feasible. To this aim, the agent may modify tasks or the goal so as to make the latter achievable. For example, the prototype allows the user to interactively revise and accept/reject changes made to task attributes to insert a new order in the initial schedule without increasing the Total Tardiness of the resulting schedule. Table 3 shows a general description of the application components.

The prototype can show graphically the evolution of the intermediate schedules in the path to the repaired schedule (and the sequential application of repair operators over the initial schedule) and learning results. Other information available to the user is the evolution of the steps per episode to reach the goal state, total reward obtained through the training phase, average steps per episode, and average reward, which is updated in real-time. The second operation mode of the prototype is "consult", that can be used once the agent has learned the repair policy. To this aim, the user can define a new schedule manually using the graphical interface, or generate it on-line in a random way, to verify the capability of the learned policy to achieve different goals from alternative initial schedules.

Table 3
General component description of the rescheduling prototype application.

Component	Description
Global Planning Application	General Global Planning System Coordinates the simulation process and presents the results graphically. Evaluates the actual state and the repair operator applied, and returns the reward. Reports if a state is a goal state
.Net Agent	Performs the repair operator application, consults the *Q*-value and abstract states, saves the states in relational format, and varies order features to make it insertable, if is not
States	Knowledge base (KB) that stores the states in relational format. KB is used to generate the RRT. It is implemented in a file called Planning.kb
RRT learner	TG Regression tree induction algorithm from ACE. Uses States, available Background Knowledge and settings from Planning.s file
Background knowledge	Set of rules that defines general knowledge across the domain. Is stored in the files Planning.bg and Background Knowledge.pl
Repair policy	Learned repair policy, reflected in the form of Prolog rules. Is parsed from PlanningQ.out to Policy.pl
Act. Abs. State	Actual Abstract State in relational format
Act. Abs. State-Repair Operator	Actual Abstract State and Repair Operator applied over it, in relational format
Consult Best Repair Operator	Wrapper Prolog embedded in OperatorManager that returns the best repair operator available, in a certain abstract state
Consult *Q*-Value	Wrapper Prolog embedded in QManager that returns the *Q*-value for a certain combination of Abstract State-Repair Operator
Abstract State Inductor	Wrapper Prolog that returns the corresponding abstract state for a particular concrete state

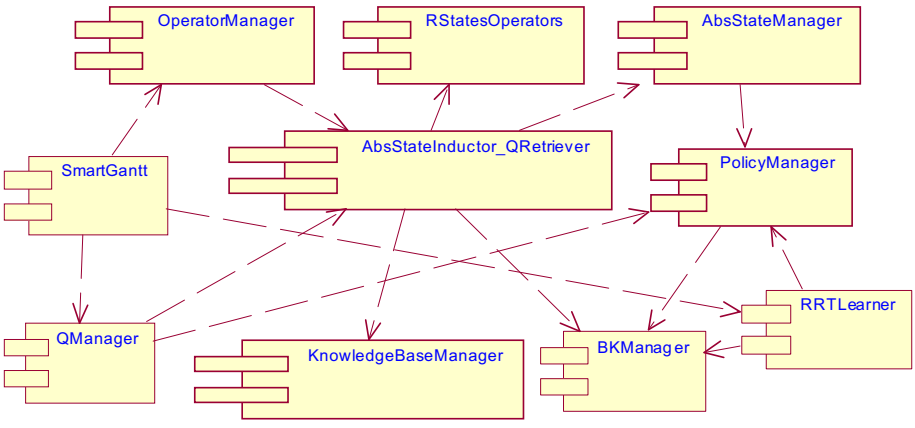


Fig. 15. *SmartGantt* component diagram.

5. Industrial case study

An example problem proposed in (Musier & Evans, 1989) is considered to illustrate the use of *SmartGantt* and 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 resource and the product being processed. For more detail, set-up times required for resource cleaning have been introduced, based on the precedence relationship between different types of final products. Processing rates and cleanout requirements are detailed in Table 4.

Order attributes correspond to product type, due date and size. In this section, this 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 the goals mentioned previously. For example, in learning to insert an order the rescheduling scenario is described by: (i) arrival of an order with given attributes that should be inserted in a randomly generated schedule state, and (ii) the arriving order attributes are also randomly chosen. This way of generating both the schedule and the new order aims to expose the rescheduling agent to sensible different situations that allow it to learn a comprehensive repair policy to successfully face the environment uncertainty. Accordingly, the initial schedule is generated in terms of the next values, which can be changed using the graphical interface of the prototype.

- *Number of orders:* the number of initial orders is determinated in a random way, using the uniform discrete distribution [10,20].
- *Order composition:* a product type is randomly selected from those available (*A*, *B*, *C* o *D*). The same probability of being selected is assigned to the four types, although this feature can be changed interactively to generate schedules containing more or fewer orders of a certain product.
- *Order size:* is obtained randomly from an interval between 100 and 1000 kg.
- Due Date: is uniformly distributed and varies between 1 and 20 days after the time of arrival.

The focal and global variables used in this example are showed in Table 5, combined with a relational representation of the schedule that has been described in previous sections. To illustrate the advantages of relational reinforcement learning in real time rescheduling, four specific situations are considered, which will be detailed in next sections. In all cases, the training is carried out with a variable number of orders in a range of 10–15, and the calculation of the average number of steps does not consider those episodes in which the agent has failed to perform a repair tasks after 1000 steps.

Table 4
A small example problem formulation (Musier & Evans, 1989).

Processing rate (lb/day)				
	A	B	C	D
Extruder #0	100	200	–	–
Extruder #1	150	–	150	300
Extruder #2	100	150	100	200
Cleanout requirements (days/cleanout)				
Previous operation	Next operation			
	A	B	C	D
A	0	0	0	2
B	1	0	1	1
C	0	1	0	0
D	0	2	0	0

Table 5
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 of all the orders in the schedule
TardinessRatio	**Global variable**. Sum over all orders in the schedule of the ration 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.** Hour in 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

5.1. Arrival of a new order

In the situation considered, there exist a certain number of orders already scheduled in the plant and a new order must be inserted so as to meet the goal state for a repaired schedule (Stability, Balancing or Tardiness Improvement). In each training episode, a random schedule state was generated, and a random insertion attempted for the new order (whose attributes are also randomly chosen), which in turn serves as the focal point for defining repair operators. Fig. 16 shows the results of the overall learning process, for each one of the available rescheduling goals. As can be seen, for Stability, the learning curve is flattened after approximately 350 episodes, when a near-optimal repair policy is obtained. For the other two more stringent situations, namely Tardiness Improvement and Balancing, learning curves tend to stabilize later, due to a higher number of repair operations that

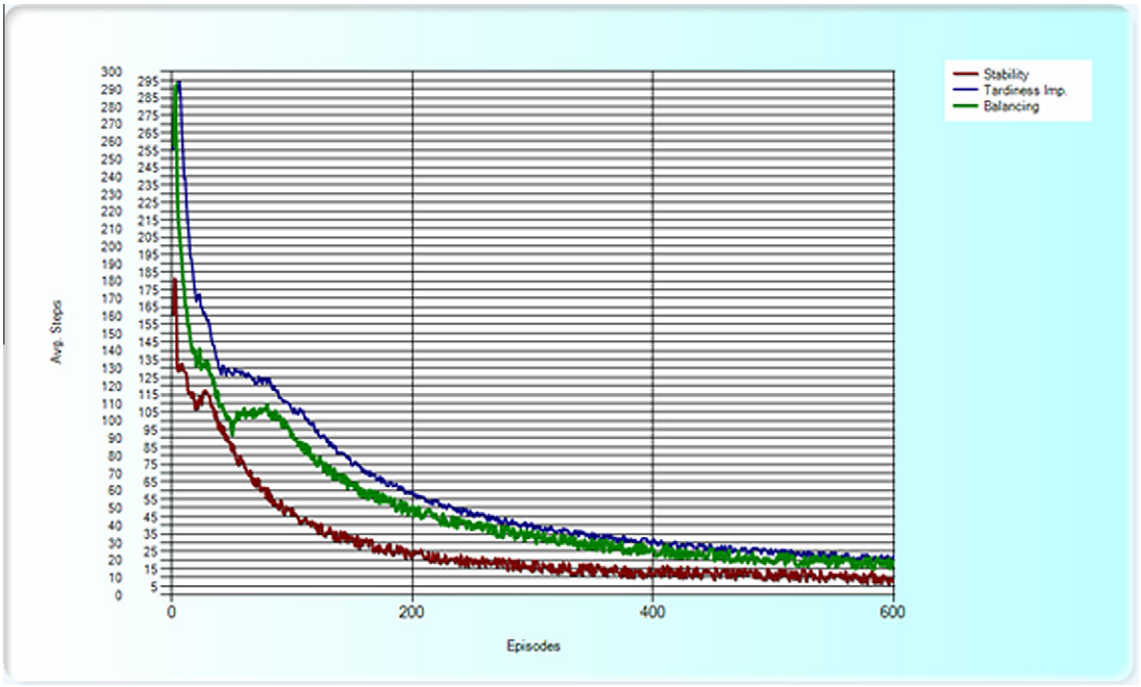


Fig. 16. Learning curve for the arrival of a new order event, with three different goals.

are necessary to try at early stages of learning so as to guarantee goal achievement.

As shown in Fig. 16, after 450 training episodes, only between 5 and 8 repair steps are required, on average, to insert a new order (regardless of the number of orders previously scheduled). Fig. 17 provides an example of applying the optimal sequence of repair operators from the schedule in Fig. 17(a), using the Consult mode of *SmartGantt*, and choosing stability as the prime objective. Before the 11th order has been included, the Total Tardiness (TT) is 19.15 h. Once the arriving order (in white) has been inserted, the Total Tardiness has been increased to 40.93 h; orange tasks are used to indicate cleaning operations. Based on the learned repair policy, a RightJump operator should be applied, which gives rise to the schedule in Fig. 17 (b) with a TT = 37.53 h. Then an UpLeftSwap repair operator is applied, decreasing TT to 25.81 TT. Then a LeftMove repair operator is used, which increases the TT to 29.42 h. Then, a DownRightSwap repair operator is used, which increases the TT to 36.25 h. After that, the applied operator is LeftSwap, decreasing TT to 22.50. Finally, by means of an UpRightSwap the goal state is reached with a Total Tardiness of 9.96 h, which is even lower than the TT in the initial schedule before the 11th order was inserted. As can be seen in the repair sequence, the policy tries to obtain an equilibrated schedule, reducing cleanout times (e.g. the cleanout operation initially presents at Ext.1), and swapping orders in order to take advantage of the resources with the best processing times. 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. As can be appreciated in Fig. 17, the number of operators that the agent must implement to achieve the goal state is rather small (6 steps). Also, although the curves tend to stabilize, a trend of gradual improvement in their behavior still can be seen.

5.2. Rush order

It is also considered here that there exist a certain number of orders already scheduled in the plant, and a new order must be inserted so as to meet the repairing goal (Balancing, Stability or Tardiness Improvement) and as an extra condition, 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. For the stability goal, a near-optimal repair policy is obtained after approximately 450 episodes. For the other two goals, the process tends to stabilize a bit later due to the tougher constraints imposed by the repairing goal (e.g. reduce the total tardiness, and at the same time respect Rush Order due date).

After 500 training episodes the rescheduling policy requires between 6 and 9 repair steps, on average, to insert the new Rush Order (Despite the number of orders previously scheduled). Fig. 18 provides a comprehensive picture of the optimal sequence of repair operators from the schedule in Fig. 18(a) when stability goal is pursued. Before the 15th order has been included, the Total Tardiness is 9.84 h. Once the arriving rush order (in white) has been inserted, the Total Tardiness has been increased to 10.90 h. Based on the learned repair policy, a RightJump operator is applied, which gives rise to the schedule in Fig. 18(b) with a TT = 12.14 h. Finally, the goal state is reached by resorting to a LeftSwap operator with a Total Tardiness of 9.77 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, move the focal task to the right in the same resource, and then swapping the position of the focal task with another located on its left."

5.3. Delay in the arrival or shortage in raw materials

Just as for the other events, in this case there exist a certain number of orders already scheduled in the plant, and an unforeseen delay in the arrival of raw materials is generated randomly for an order, which is in turn taken as focal point. This order must be reprogrammed so as to complain with the new restriction, and to meet the chosen rescheduling goal. Training has been performed in a similar manner as for the other events. For stability, a near-optimal repair policy is obtained after approximately 450 episodes. For the other two goals, the process tend to converge later, due to the restrictive conditions established by the nature of the

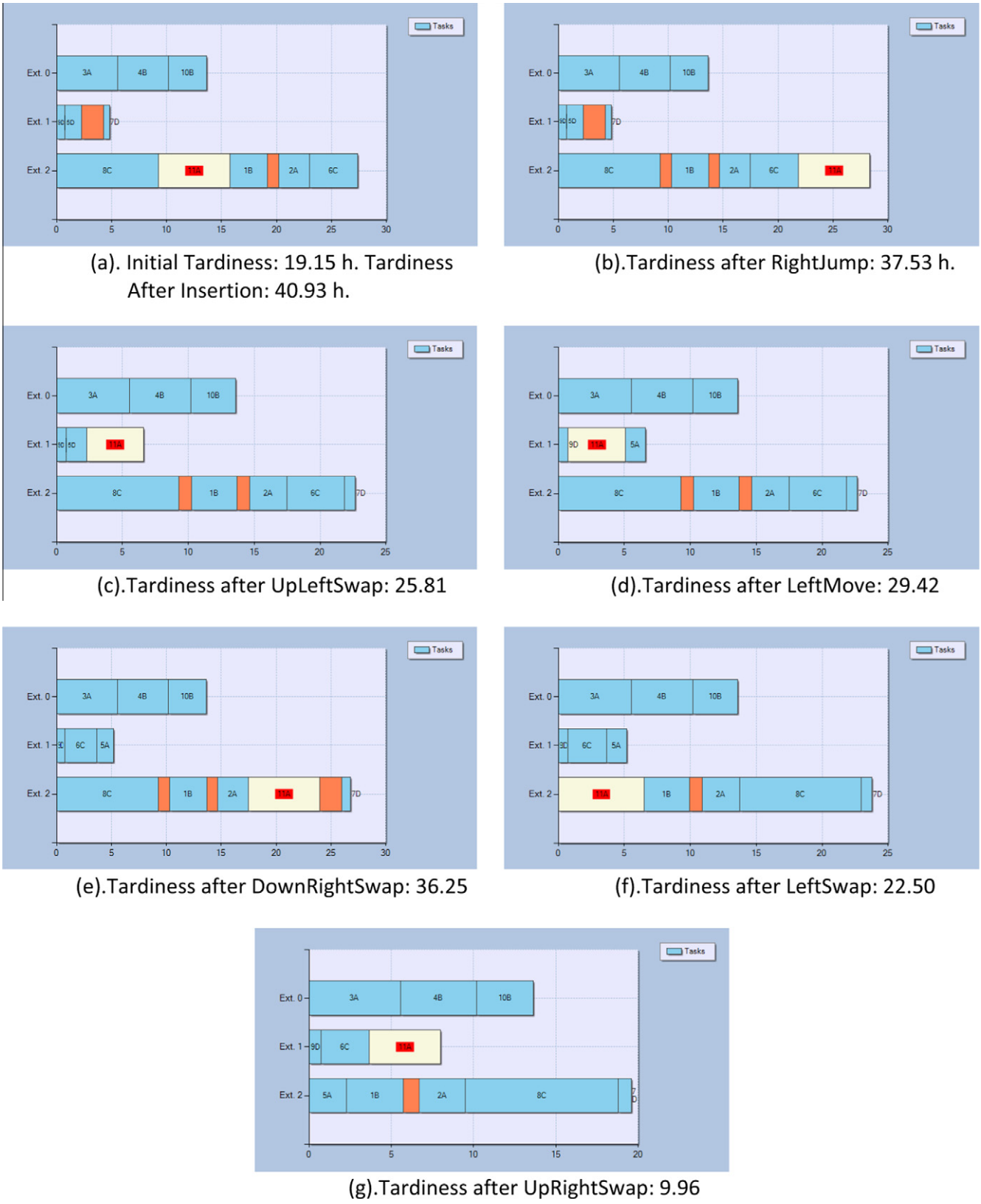
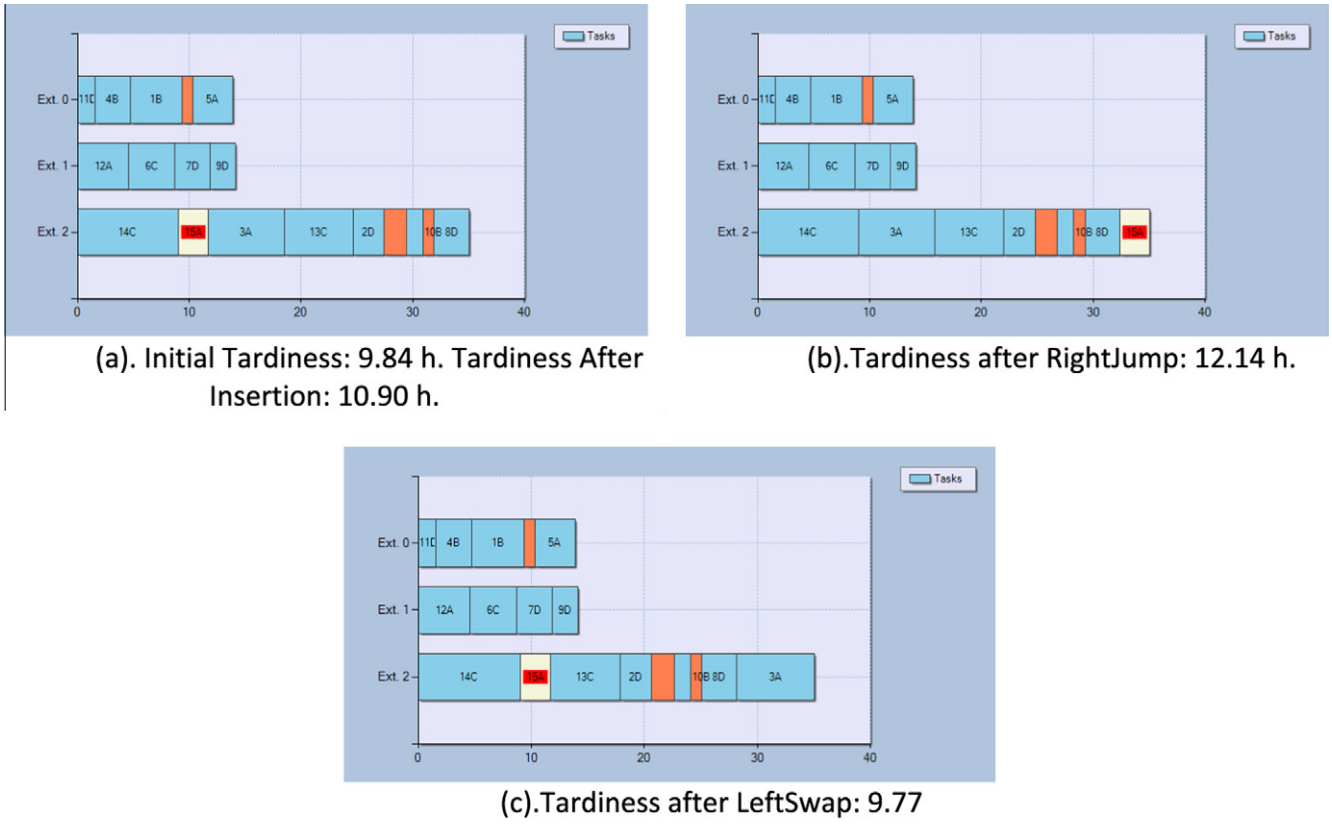


Fig. 17. Repairing sequence for arrival of a new order event.

rescheduling goals and the number of repair operations that are required to reach the goal state (e.g. balancing tardiness reduction with the number of required steps to achieve the goal, and at the same time reprogramming the order to accomplish with the delayed arrival of raw materials).

After approximately 500 training episodes, between 7 and 9 repair steps are required, on average, to reschedule the affected order (regardless of the number of orders previously scheduled). Fig. 19 provides an example of applying the optimal sequence of repair operators from the schedule in Fig. 19(a), using the Consult mode

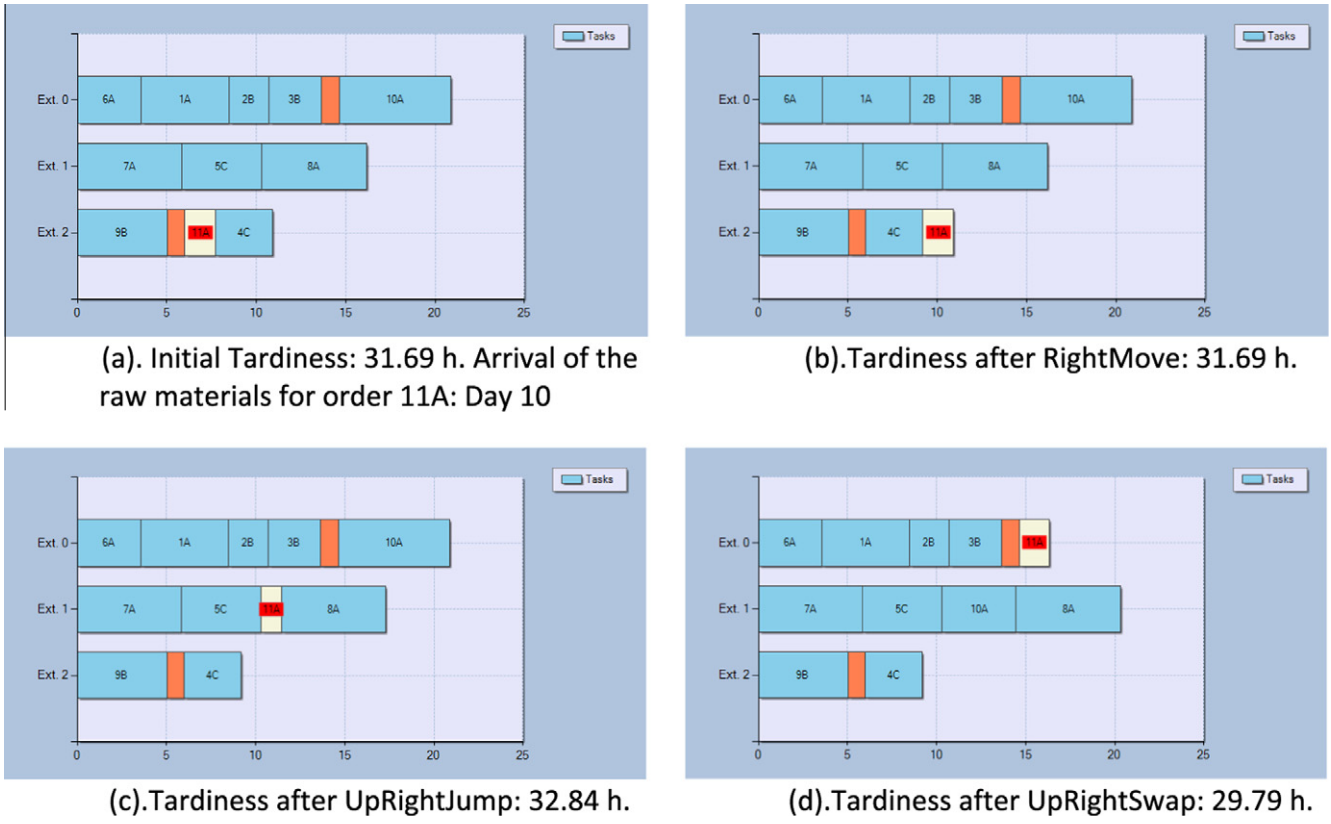


(a). Initial Tardiness: 9.84 h. Tardiness After Insertion: 10.90 h.

(b).Tardiness after RightJump: 12.14 h.

(c).Tardiness after LeftSwap: 9.77

Fig. 18. Repairing sequence for rush order event.



(a). Initial Tardiness: 31.69 h. Arrival of the raw materials for order 11A: Day 10

(b).Tardiness after RightMove: 31.69 h.

(c).Tardiness after UpRightJump: 32.84 h.

(d).Tardiness after UpRightSwap: 29.79 h.

Fig. 19. Repairing sequence for delay in the arrival or shortage in raw materials event.

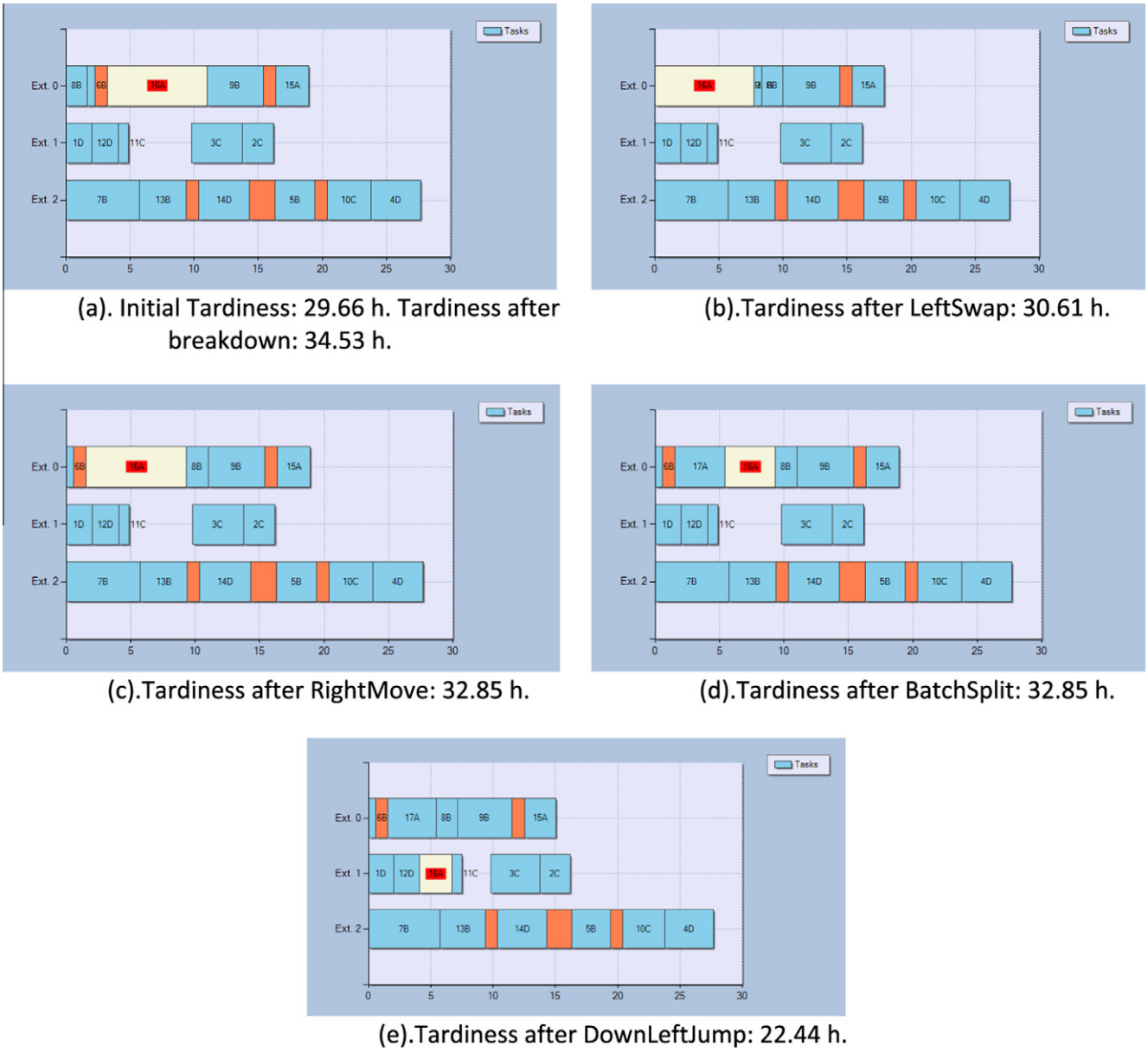


Fig. 20. Repairing sequence for machine breakdown event.

of *SmartGantt*, and pursuing the stability goal. Before the delay occurrence, there exist 11 orders already scheduled in the system, and the Total Tardiness is 31.69 h. Then, a delay of 4 days in the arrival of raw materials is produced for the order #11 related to product A. Based on the learned repair policy, *SmartGantt* applies a RightJump operator, trying to delay the start time of Order #11, which gives rise to the schedule in Fig. 19(b) with a TT = 31.69 h. Despite tardiness is not yet increased, the new start time for order #11 is not satisfactory, because it still begins before the planned arrival of raw material. So, the system applies the UpRightJump operator, trying to change the assigned resource for the order and at the same time, delay its start time. This operation gives rise to the schedule shown in Fig. 19(c), with a TT = 32.84. Although order #11 is now schedule 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 initial. At this point, the user must tell the system whether to continue with repairing process, or maintain the current schedule. If the user decides to continue, *SmartGantt* applies the UpRightSwap operator, changing the position of order #11 with order #10. The goal state is then reached with a Total Tardiness of 29.79 h, which is lower than the TT in the initial schedule.

he relational heuristic learned through intensive simulation, and applied by *SmartGantt* by means of Prolog rules, in this case (which can be different for other schedule initial conditions and configurations) shows vividly how the use of the relational-deictic representation allows that the policy becomes meaningful for the operator, and can be synthesized, for example, through the next statement: "if the start time of the affected order is lower than the new date for raw material availability, try to move the order 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 are made to the relative position of other scheduled orders. If the rescheduling goal has not still been reached, then try a more complex operator to produce a higher delay in the start time of the order but involving a swap operation with a task scheduled in an alternative resource."

5.4. Machine breakdown

The last disruptive event that has been considered in this work is machine breakdown. For simulation purposes, is assumed that there exist a certain number of orders already scheduled in the plant, and an unexpected breakdown for a limited amount of time is generated randomly for a given resource. Then, the system must select a task to take as focal point, and then beginning solving the rescheduling task. For this event, in a great number of episodes *SmartGantt* must relax the conditions stated for the goal state, particularly in those cases where the user selects Tardiness Improvement or Balancing as goals. For Stability, a near-optimal repair policy is obtained after approximately 700 episodes. For the other two goals, the process only converges after 800–900 training episodes, possibly due to the very restrictive conditions established by the nature of repair schedules that are expect and the number of operations needed to reach the chosen goal state.

After approximately 700 training episodes, on average 10 repair steps are required to perform the rescheduling task. Fig. 20 provides an example of applying the optimal sequence of repair operators starting from the schedule in Fig. 20(a) and having chosen the stability goal. Before the resource breakdown event occurs, there exist 16 orders already scheduled in the system, and the Total Tardiness is 29.66 h. Then, a breakdown of 2 days in the extruder #2 is generated, which gives rise to the schedule in Fig. 20(a) with a TT = 34.53 h. Based on the learned repair policy, *SmartGantt* select as focal the order #16, and applies a LeftSwap operator, trying to reduce the total tardiness, which is decreased to 30.61 h. as can be seen in Fig. 20(b). Then, *SmartGantt* applies a RightMove operator, increasing TT to 32.85 h. The goal state has not been reached, so the rescheduling system applies a BatchSplit operator -which does not increases tardiness-, and finally the goal is reached using the DownLeftJump operator, which reduces the TT to 22.44 h, as is shown in Fig. 20(e). It is important to highlight the usefulness of BatchSplit operator in learning of the repair policy for a resource breakdown, which allows the system to exploit small idle times left in all resources.

6. Concluding remarks

A novel approach and a prototype application for – called *SmartGantt* – simulation-based learning of a relational policy dealing with automatic repair in real time of plans and schedules based on relational reinforcement learning has been proposed. The policy allows generation of a sequence of deictic (local) repair operators to achieve several rescheduling goals to handle abnormal and unplanned events such as inserting an arriving order with minimum tardiness based on a relational (deictic) representation of abstract schedule states, and repair operators. Representing schedule states using a relational (deictic) abstraction is not only efficient to profit from, but also potentially 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. These repair policies relies on abstract states, which are induced for generalizing and abstracting ground examples of schedules, allowing the use of a compact representation of the rescheduling problem. Abstract schedule states and repair actions facilitates and accelerates learning and knowledge transferring, which is independent of the type of event that has generated a disruption and can be used reactively in real-time. An additional advantage provided by the relational (deictic) representation of schedule (abstract) states and operators is that, relying in an appropriate and well designed set of background knowledge rules, it enables the automatic generation through inductive logic programming of heuristics that can be naturally understood by the end-user, and facilitates tasks like "what–if" analysis, interactive rescheduling and decision support.

Scale-up reinforcement learning using relational modeling of rescheduling situations based on simulated transitions is a very appealing approach to compile a vast amount of knowledge about repair policies, where different types of abnormal events (order insertion, extruder failure, rush orders, reprocessing needed, etc.) can be generated separately and then compiled in the relational regression tree, regardless of the event used to generate the examples (triplets). This is another very appealing advantage of the proposed approach, since the repair policy can be used online to handle disruptive events that are even different from the ones used to generate the *Q*-function; once this function is known, reactive scheduling is straightforward. Finally, relational reinforcement learning favors denotational concept semantics in the communication between humans and *SmartGantt*. This capability of phrasing the rationale behind repair strategies so as to receive evaluative feedback verbally from human users is obviously highly useful in real-world applications of *SmartGantt*.

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*(2), 397–422.

Aytug, H., Lawley, M., 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*, 86–110.

Blockeel, H., & De Raedt, L. (1998). Top-down induction of first order logical decision trees. *Artificial Intelligence, 101*(1–2), 285–297.

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*(1), 59–93.

Chieh-Sen, H., Yi-Chen, H., & Peng-Jen, L. (2012). Modified genetic algorithms for solving fuzzy flow shop scheduling problems and their implementation with CUDA. *Expert Systems with Applications, 39*, 4999–5005.

Croonenborghs, T. (2009). *Model-assisted approaches to relational reinforcement learning*. Ph.D. Dissertation. Leuven, Belgium: Department of Computer Science, Katholieke Universiteit Leuven.

Croonenborghs, T., Driessens, K., & Bruynooghe, M. (2008). Learning relational options for inductive transfer in relational reinforcement learning. In *Seventeenth international conference on inductive logic programming (ILP) 2007. LNCS* (Vol. 4894, pp. 88–97). Springer.

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

De Raedt, L., & Džeroski, S. (1994). First order *jk*-clausal theories are PAC-learnable. *Artificial Intelligence, 70*, 375–392.

Driessens, K. (2004). *Relational reinforcement learning*. Ph.D. Dissertation. Leuven, Belgium: Department of Computer Science, Katholieke Universiteit Leuven.

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.). *Twelfth European conference on machine learning, ECML 2001, Freiburg, Germany, September 5–7, LNCS* (Vol. 2167, pp. 97–108). Springer.

Džeroski, S., De Raedt, L., & Driessens, K. (2001). Relational reinforcement learning. *Machine Learning, 43*(1–2), 7–52.

Gärtner, T. (2008). *Kernels for structured data. Series in machine perception and artificial intelligence* (Vol. 72). Singapore: World Scientific Publishing.

Henning, G. (2009). Production scheduling in the process industries: Current trends, emerging challenges and opportunities. *Computer-Aided Chemical Engineering, 27*, 23–36.

Henning, G., & Cerdá, J. (2000). Knowledge-based predictive and reactive scheduling in industrial environments. *Computers and Chemical Engineering, 24*, 2315–2338.

Herroelen, W., & Leus, R. (2004). Robust and reactive project scheduling: A review and classification of procedures. *International Journal of Production Research, 42*(8), 1599–1620.

Li, Z., & Ierapetritou, M. (2008). Reactive scheduling using parametric programming. *AIChE Journal, 54*(10), 2610–2623.

Martinez, E. (1999). Solving batch process scheduling/planning tasks using reinforcement learning. *Computers and Chemical Engineering, 23*, S527–S530.

McKay, K. N., & Wiers, V. C. S. (2001). *Decision support for production scheduling tasks in shops with much uncertainty and little autonomous flexibility*. *Human performance in planning and scheduling*. New York: Taylor and Francis.

Méndez, C., Cerdá, J., Harjunkoski, I., Grossmann, I., & Fahl, M. (2006). State-of-the-art review of optimization methods for short-term scheduling of batch processes. *Computers and Chemical Engineering, 30*, 913–946.

Mitchell, T. (1997). *Machine learning*. New York: MacGraw Hill.

Miyashita, K. (2000). Learning scheduling control through reinforcements. *International Transactions in Operational Research, 7*(2), 125–138.

Miyashita, K., & Sycara, K. (1995). CABINS: A framework of knowledge acquisition and iterative revision for schedule improvement and iterative repair. *Artificial Intelligence, 76*(1–2), 377–426.

Morales, E. F. (2003). Scaling up reinforcement learning with a relational representation. In *Proceedings of the workshop on adaptability in multi-agent systems at AORC'03, Sydney, Australia*.

Morales, E. F. (2004). Relational state abstraction for reinforcement learning. In *Proceedings of the twenty-first international conference (ICML 2004), July 4–8, Banff, Alberta, Canada*.

Musier, R., & Evans, L. (1989). An approximate method for the production scheduling of industrial batch processes with parallel units. *Computers and Chemical Engineering, 13*, 229–238.

Palombarini, J., & Martínez, E. (2010). Learning to repair plans and schedules using a relational (deictic) representation. *Brazilian Journal of Chemical Engineering, 27*(03), 413–427.

Pinedo, M. L. (2005). *Planning and scheduling in manufacturing and services*. New York: Springer.

Pinedo, M. L. (2008). *Scheduling: Theory, algorithms, and systems* (3rd ed.). New York: Springer.

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

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

Torrey, L., Shavlik, J., Walker, T., & Maclin, R. (2006). Skill acquisition via transfer learning and advice taking. In *Seventeenth European conference on machine learning* (pp. 425–436).

Trentesaux, D. (2009). Distributed control of production systems. *Engineering Applications of Artificial Intelligence, 22*, 971–978.

Van Otterlo, M. (2009). *The logic of adaptive behavior: Knowledge representation and algorithms for adaptive sequential decision making under uncertainty in first-order and relational domains*. Amsterdam: IOS Press.

Vieira, G., Herrmann, J., & Lin, E. (2003). Rescheduling manufacturing systems: A framework of strategies, policies and methods. *Journal of Scheduling, 6*, 39–62.

Yagmahan, B., & Yenisey, M. M. (2010). A multi-objective ant colony system algorithm for flow shop scheduling problem. *Expert Systems with Applications, 37*, 1361–1368.

Zaeh, M., & Ostgathe, M. (2009). A multi-agent-supported, product-based production control. In *Proceedings of the seventh IEEE international conference on control and automation, ICCA 2009* (pp. 2376–2383).

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*, 300–307.

Zhang, W., & Dietterich, T. (1995). Value function approximations and job-shop scheduling. In J. A. Boyan, A. W. Moore, & R. S. Sutton (Eds.), *Proceedings of the workshop on value function approximation*. Report number CMU-CS-95-206. Carnegie-Mellon University, School of Computer Science.

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*, 1588–1596.