Real-time Rescheduling of Production Systems using Relational Reinforcement Learning

Jorge Palombarini[1], Ernesto Martinez[2]

[1] GISIQ - UTN - Fac. Reg. V. María, Av. Universidad 450,
5900, Villa María, Argentina.
jpalombarini@frvm.utn.edu.ar
[2] INGAR(CONICET-UTN), Avellaneda 3657,
S3002 GJC, Santa Fe, Argentina.
ecmarti@santafe-conicet.gob.ar

Abstract. Most scheduling methodologies developed until now have laid down good theoretical foundations, but there is still the need for real-time rescheduling methods that can work effectively in disruption management. In this work, a novel approach for automatic generation of rescheduling knowledge using Relational Reinforcement Learning (RRL) is presented. Relational representations of schedule states and repair operators enable to encode in a compact way and use in real-time rescheduling knowledge learned through intensive simulations of state transitions. An industrial example where a current schedule must be repaired following the arrival of a new order is discussed using a prototype application –*SmartGantt*®- for interactive rescheduling in a reactive way. *SmartGantt*® demonstrates the advantages of resorting to RRL and abstract states for real-time rescheduling. A small number of training episodes are required to define a repair policy which can handle on the fly events such as order insertion, resource break-down, raw material delay or shortage and rush order arrivals using a sequence of operators to achieve a selected goal.

Keywords: Learning, Rescheduling, Relational modeling, agile manufacturing.

1 Introduction

The scheduling task can be described as assigning a set of activities to a limited number of resources in a consistent manner over time, so as to avoid violations of the restrictions associated with the problem, such as limited resources capacity, due dates, or precedence relations between activities, while a determined set of objectives is optimized [1].

Although this problem has been extensively studied, most of the existing works addressing schedule optimization are based on the assumptions of complete information and a static and fully deterministic environment [2]. A pervasive assumption in this conception has been that the optimized schedule, once released to the production floor, can be executed as planned. Nevertheless, a schedule is typically subject to the intrinsic variability of a process environment where difficult-to-predict

events occur as soon as it is released for execution, like equipment failures, quality tests demanding reprocessing operations, arrival of rush orders and delays in material inputs from previous operations, and continued advent of new orders, which increase the associated uncertainty in real time schedule execution. So, elaborated plans fast become obsolete, and they need to be updated constantly [3]. Hence, the inability of most scheduling literature to address the general issue of uncertainty and variability is often cited as a major reason for the lack of influence of current research in the field on industrial practice [4].

Real-time rescheduling (or reactive scheduling) is the knowledge-intensive activity of updating an existing schedule in response to disturbances and unplanned events [1], and is a key issue in disruption management. Existing approaches for generating (near) optimal schedules for real-world production systems, typically require significantly large amounts of time, due to the inherent computational complexity of this type of problems, and the uncertainty related to the environment dynamics, so the cost for searching an optimal schedule is prohibitively high. If repair decisions need to be made in real-time, then fast rescheduling is mandatory to account for unplanned and abnormal events by generating satisfying schedules rather than optimal ones [1]

Therefore, the capability of generating and representing knowledge about heuristics for repair-based scheduling is a key issue in any rescheduling strategy, and many works have been developed exploiting peculiarities of the specific problem structure ([5],[6],[7],[8]), but the tricky issue is that resorting to a feature-based representation of schedule state is very inefficient, and generalization to unseen states is highly unreliable, as well as the learning performed and the acquired knowledge are difficult to transfer to unseen scheduling domains. In contrast, humans can succeed in rescheduling thousands of tasks and resources by increasingly learning 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. First-order relational representations enable the exploitation of the existence of domain objects, of relations (or, properties) over these objects, and enable the use of *quantification* over objectives (goals), action effects and properties of states.

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 using simulations [9], implements an interactive scheduling repair bearing in mind different goals and scenarios. To this aim, domain-specific heuristics for scheduling repair are developed using two general-purpose algorithms already available: TILDE and TG ([10],[11]).

2 *SmartGantt®* repair-based (re)scheduling

Fig. 1 depicts the repair-based architecture implemented by *SmartGantt®*, where search control knowledge about repair operator optimal selection is acquired through reinforcements using a schedule state simulator. In the simulation environment, an instance of the schedule is interactively modified by the system using a sequence of repair operators until a repair goal is achieved or the impossibility of repairing the schedule is accepted. In each learning episode , *SmartGantt®*, receives information

from the schedule situation or state *s* and then selects a repair operator *a* to be applied to the current schedule, resulting in a new one. The evaluation of resulting quality of a schedule after the 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 [12] such as the well-known *Q*-learning rule. 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 issue for reinforcement learning is then how schedules states and repair strategies must be represented for knowledge acquisition and iterative revision. 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 are no extra burden on domain experts, online adaptation to a dynamic environment, and possibility of incorporating abstractions to deal with large state spaces.

.

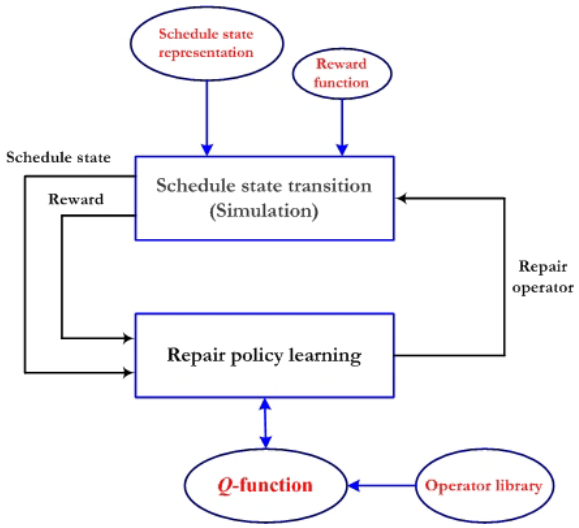

Fig. 1. Repair Based Rescheduling

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, e. g. states where total 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 δ is the transition function which is unknown to the learning agent.

Also, a reward function is used as guideline to learn a repair policy from reinforcements based on simulations [13]. Resorting to the reward function and simulations, the optimal *policy*: $a_i = \pi^*(s_i)$ can be approximated using a regression tree ([14],[15]), and can be used to compute the shortest action-sequence to reach a

repaired scheduled, improving responsiveness at the shop-floor to handle unplanned events and meaningful disturbances at the shop-floor [23].

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 [10], and offers many possibilities for abstraction due to the structured form of ground atoms in the 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 ([11],[14]). This is usually characterized by a large and possibly unbounded number of different states and actions as the case of planning and scheduling. Rather than using an explicit state-action *Q*-table like traditional reinforcement learning algorithms, RRL stores the *Q*-values in a logical regression tree [15]. The relational version of the *Q*-learning algorithm is shown in Fig. 2.

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, the fact that the learner is given a continuous stream of (*state*, *action*, *q-value)*-triplets and has to predict *q-values* for (*state*, *action*)-pairs during learning.

```
Initialize the Q-function hypothesis Q̂_0
e ← 0
repeat
 Examples ← ∅
 Generate a starting schedule state s_0
 i ← 0
 repeat
 choose a repair operator a_i at s_i using a policy (e.g., ε-greedy)
based on the current
 hypothesis Q̂_e implement operator a_i, observe r_i and the resulting
schedule s_{i+1}
 i ← 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, q̂_j), where q̂_j ← r_j + γ max_a Q̂_e(s_{j+1}, a)
 Examples ← Examples ∪ {x}
 end for
 Update Q̂_e to Q̂_{e+1} using Examples and a relational regression
```

Fig. 2. A RRL algorithm for learning to repair schedules through intensive simulations.

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 ([12],[10]). During each learning episode, all the encountered states and the selected actions are stored, together with the rewards related to each encountered (*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 then presents the set of (*state*, *action*, *qvalue*)-triplets to a relational regression engine, which will use this set of *Examples* to update the current regression tree of the *Q*-function, and then the algorithm continues executing the next learning episode.

Several incremental relational regression techniques have been developed to meet the requirements for RRL implementation: an incremental relational tree learner TG [16], an instance based learner [17], a kernel-based method ([18],[20]) and a combination of a decision tree learner with an instance based learner [19].
TG is the most popular relational regression algorithm that has been developed for policy representation and state abstraction in logical and relational learning ([10],[11],[14]), and 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*. The difference with traditional decision tree learners is in the generation of the tests to be incorporated in the nodes, due to the symbolic nature of representation used for states of schedule and repair operators. For this, the algorithm employs a refinement operator ρ that works under θ-*subsumption*, based on information gain [11].

3.1 Relational (deictic) Representation of Schedule States and Repair Operators

The problems of attribute-value representations that we have described in previous sections, are solved by *SmartGantt®* looking at *relational* (or *first-order*) deictic representations. Such a relation, presupposes a *language* to express sets of *relational facts* that describe a schedule state and available repair operators in a compact and comprised way; each state can be characterized by only those facts that hold in it, that are obtained applying a *hold(State)* function.

Such application, implements the concept of "abstracting (learning) from interpretations" [21]. In this notation, each (state, action) pair will be represented as a set of relational facts, which is called a relational interpretation. Additionally, it is proposed here a deictic approach to define relational schedule states and repair operators as a much powerful alternative used by *SmartGantt®*, which deals with the varying number of tasks in the planning world by defining a focal point for referencing objects in the schedule state.

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 reactor, and iii) Tasks related to the last order.

Fig. 3 shows a relational (deictic) representation of a schedule. Note that the number of facts in an example is not fixed, and that the order of the facts is arbitrary. For example, the structure of `precedes`(task1,task2) shares the parameter object "task2" with the fact `precedes`(task2,task3). This type of structural pattern can be

exploited by SmartGantt for compact abstractions, something which is not possible in a propositional representation of schedule states.

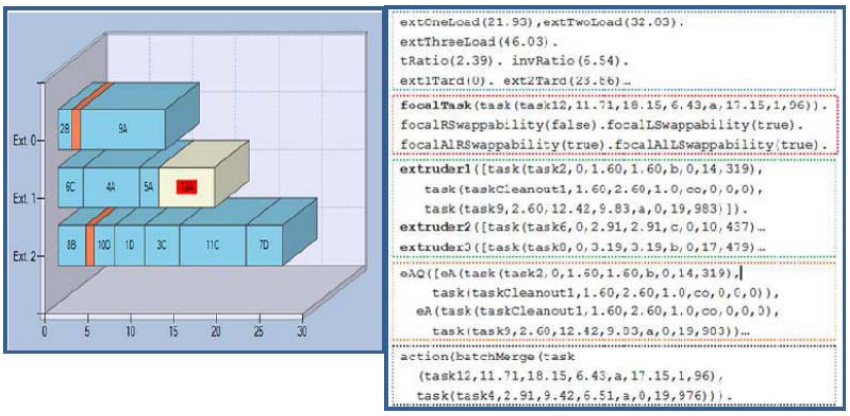

Fig. 3. Relational representation of a schedule state

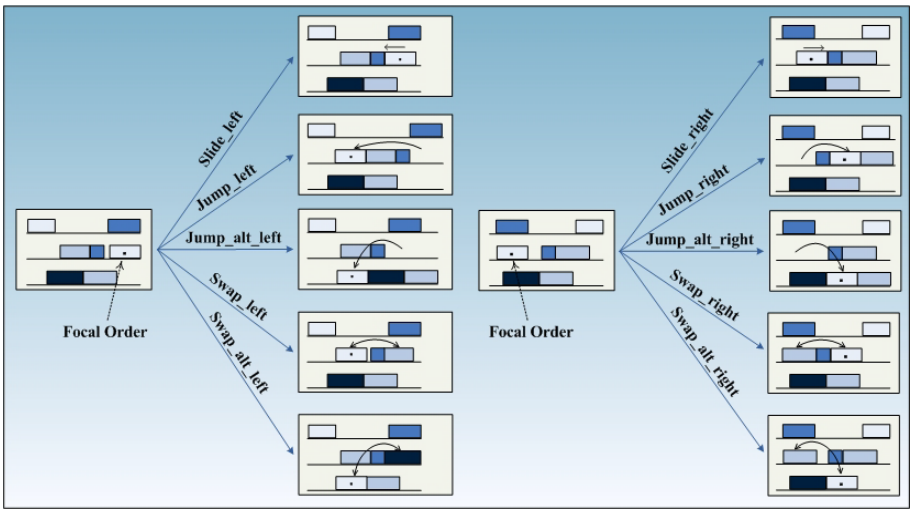

Fig. 4. Deictic Repair Operators

In a deictic representation, both scheduling states and repair operators (actions) are defined in relation to a given focal point (i.e., a task) as shown in Fig. 4. 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 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.

3.2 Inducing Abstract Schedule States and Repair Policy Using Logical Decision Trees

As we have seen previously, relational representations of the schedule states and repair operators are *symbolic* in nature; so, the main challenge in this type of domains is to exploit the inherent structure of schedule states, as well as the structure shared among several schedules or parts of them using *variables* to range over constants, and Background Knowledge (BK) with syntactic bias in the form of types and modes, which consists of the definitions of general predicates that can be used in the induced hypotheses, for generalization, abstraction, and knowledge transferring purposes.

To this aim, and to reduce the state space and accelerate learning, *SmartGantt®* performs an induction process over the schedule states and repair operators, obtaining a set of **Abstract States** (AS), which are conjunctions $\equiv 1 \wedge \ldots \wedge m$ of logical atoms, (e.g., a logical query).

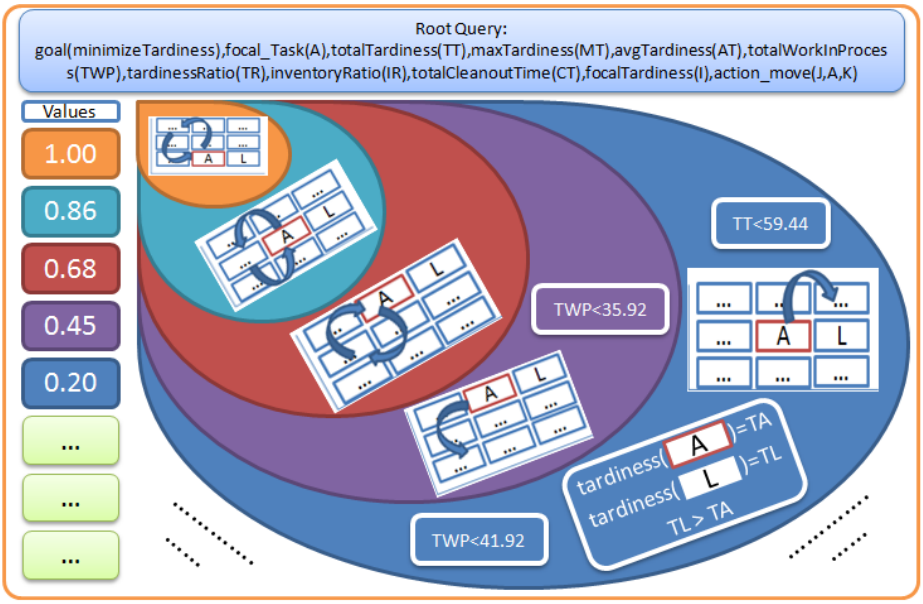

Fig. 5. Part of the Abstract State Value Function for the minimize tardiness goal.

The use of variables, as shown in Fig. 5 in the Root Query, admits abstracting over specific domain objects as well. The application starts with a first-order logical alphabet containing predicates, constants and variables. Thus, an AS is basically a logical sentence, specifying general properties of several states visited during the learning through simulated transitions. An example can be the AS depicted in Fig. 5, where 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 the last, the total Work in Process is less than 41.92 h., and the repair operator that has been applied is `JumpAltRight`. Note that the real task denoted by A must be the same in all cases where such variable appears. So, an AS does not depend on specific objects like task1, extruder1, but refers to them through variables.

Fig. 5 shows that the action-value function relies on a set of abstract states, which together, encode the kind of rescheduling knowledge learned through intensive simulation in a compact and comprised way, which 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: it only depends on the desired goal for the repaired schedule state. As a result, it is no important to determinate the event type that has driven the schedule to the current state, to find a sequence of repair operators to achieve a goal for the resulting schedule. Using this powerful abstraction, the states are characterized by a set of properties, rather than a particular way, and the repair policy express the relative position of the learning system with respect to other objects in the schedule domain, which also allows it to transfer the learned policies to different problems were the same relations apply, without any further learning. An abstract state *S,* covers a ground state *s* iff *s* /= *S*, which is decided by *SmartGantt®* using *θ-subsumption* [17].

The task of inducing the set of abstract states is carried on by *SmartGantt®* using the result of applying TG, in combination with the algorithm depicted in Fig. 6 that was implemented in Prolog.

```
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. 6. Algorithm that determines the Abstract State corresponding to a ground state.

Encoded rescheduling knowledge based on abstract states is used each time the prototype needs to consult the *Q*-value of a certain *state-repair operator* pair (*s,a)*. To this aim, using the collection of rules derived by TG, it stores the body (set of logical sub-objectives that define an abstract state) of each rule, in a separate collection of Abstract States (ABS). Then, it loads (consults) the available Background Knowledge, and ABS; for each element in ABS, it checks the *θ-subsumption* between the particular ground state-repair operator (s,a), and the abstract state. If it succeeds, then the algorithm returns the corresponding abstract state.

In the RRL approach just presented, the learner generates the definition of the *Q*-function from a set of examples in the form of abstract state-action-value tuples, and dynamically partitions the set of possible states (different regions in Fig. 5).

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. **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. 7, and Fig. 8 shows its graphical user interface.

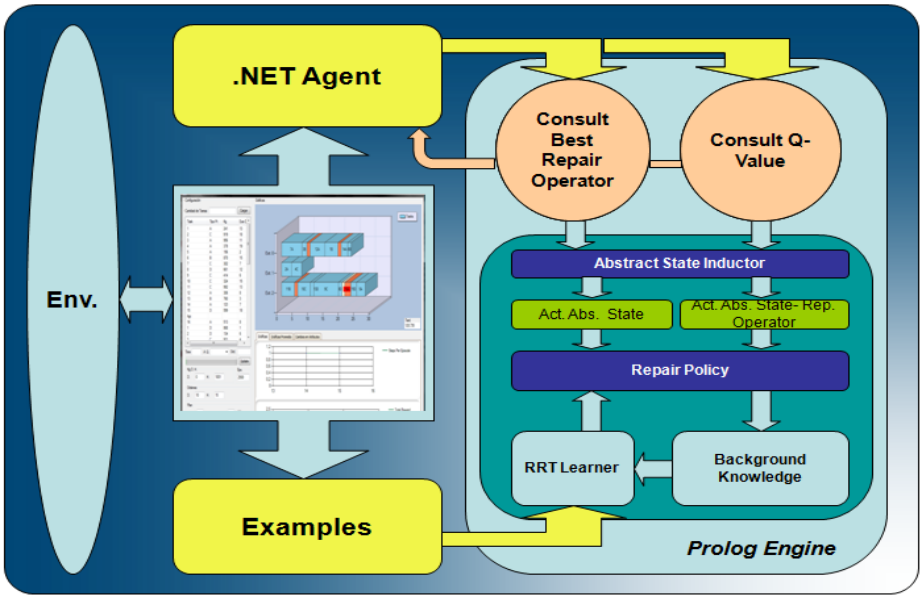

Fig. 7. Graphical schema of the SmartGantt architecture

The prototype admits two 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, and encodes the *Q*-function so that it can then apply this knowledge in the consult phase. As an example, the disruptive event that the system can handle is the arrival of a new order to the production system is considered.

Before starting the training phase, using the graphical interface, the user must define the value of the simulation and training parameters, related to:

- *Initial schedule conditions*: to the 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*: the user must adjust the RRL associated parameters, like γ, ε, and α.
- *Goal State Definition*: the last parameter that has to be set before starting simulation, is the desired goal in case that the disruptive event is the arrival of a new order to the production system. Checking the option "Try Reinsertion",

enables the agent 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, in the case that the order cannot be inserted with its initial requirements.

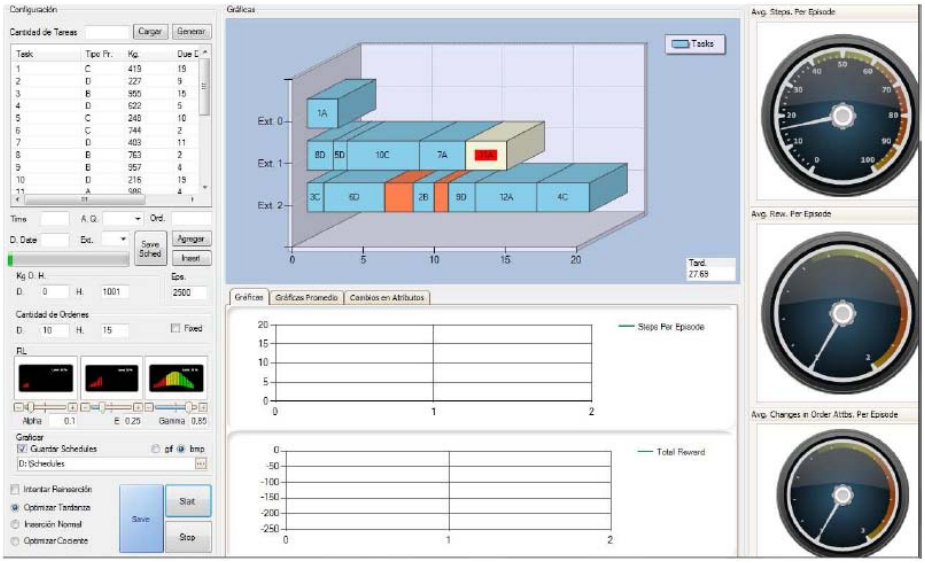

Fig. 8. SmartGantt Graphical User Interface

Training a rescheduling agent can be carried out having three possible goals in mind, which can be selected through an option list:

- Tardiness Optimization: the repaired schedule has less tardiness than the initial one.
- Normal Insertion: tries to minimize the number of moves, and the reward r is assigned as follows: $r=(n_0-n_c)/n_0$, if $goal(s_t)=true$ and $r=0$, if $goal(s_t)=false$. n_0 is the total number of orders, and n_c is the number of orders that have changed their position with respect to the original schedule.
- Quotient Optimization: tries to optimize the tardiness T and the number of changes made to the original schedule N, as follows: $r=(TInitial-TFinal)/N$. 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 the reward is assigned to the applied operators. At the beginning of each training episode, the prototype generates a set of orders with their corresponding attributes, bounded over allowable ranges 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 it is assigned arbitrarily to one of the extruders. The learning episode progresses by applying a sequence of repair operators until the goal state is reached.

Based on the suitable initialization of the *Q*-function, the RRL algorithm starts to experiment with simulated learning episodes which allows 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. So, the algorithm gives the set of tuples (state, action, *Q*) 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 generated tree, different nodes are basically Prolog queries. In consequence, to find a *Q*-value, a Prolog program based on the tree is built on-line. 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 rely upon defined rules of the background knowledge, which is valid all over the domain.

The relational regression tree contains (in relational format) the repair policy learned from the current and previous episodes. The mentioned queries are actually processed by the Prolog wrappers **ConsultBestAction.exe** and **ConsultQ.exe,** which made up a transparent interface between the .NET agent and the relational repair policy. 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. 6 is implemented in a separated library that uses the functionality of Prolog.NET to perform the induction of abstract states.

Table 1. General component description of the rescheduling prototype application.

Component	Description
Environment	Evaluates the pair state-repair operator, and returns the reward. Reports if a state is a goal state.
Application	Coordinates the simulation process and presents the results graphically.
.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.
Examples	Contains the states in relational format, and is used to generate the RRT.
RRT Learner	Regression Tree induction algorithm.
Background Knowledge	Set of rules that defines general knowledge of the domain.
Repair Policy	Learned repair policy, reflected in the form of Prolog rules.
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 that returns the best repair operator available, in a certain Abstract State-Goal.
Consult Q Value	Wrapper Prolog that returns the Q value for a certain combination of Abstract State-Repair Operator-Goal
Abstract State Inductor	Wrapper Prolog that returns the corresponding abstract state for a particular concrete state.

In the .NET prototype, different classes are used to model Agent, Environment, Actions and Policy using the files Policy.pl, ActState.pl, ActStateAction.pl and BackgroundKnowledge.pl. Finally, the .NET agent is fully equipped to handle situations where the order 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. The prototype allows the user to interactively revise and accept/reject changes made to order attributes so as to insert it in the initial schedule without increasing the Total Tardiness of the resulting schedule.

The prototype can show graphically the evolution of the insertion (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 (changes made to original schedule) 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 random way, to verify the behavior of learned policy.

5 Example

An example problem proposed by Musier and Evans in [22] is considered 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 resource and the product being processed. For resource set-up, times required to resource cleaning have been introduced, based on the precedence relationship between types of elaborated products. Processing rates and cleanout requirements are detailed in [23]. 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 not increase the total tardiness when a new order needs to be inserted. In learning to insert an order the situation before the sequence of repair operations is applied 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 exposes the agent to totally dissimilar situations learning a repair policy to successfully face the environment uncertainty.

So, the initial schedule is generated in terms of several values, which can be changed using the graphical interface of the prototype, such as number of orders, order composition (product types), order size and due dates.

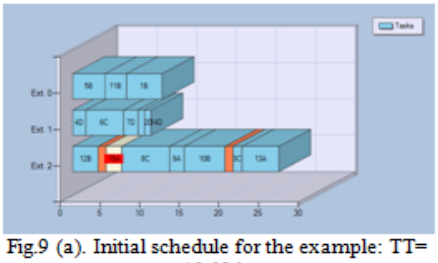

Fig.9 (a). Initial schedule for the example: TT= 45.52 h.

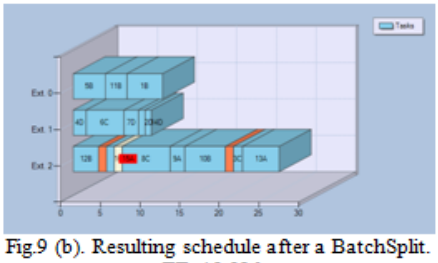

Fig.9 (b). Resulting schedule after a BatchSplit. TT=45.52 h.

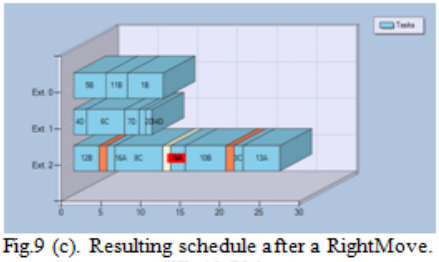

Fig.9 (c). Resulting schedule after a RightMove. TT=49.73 h.

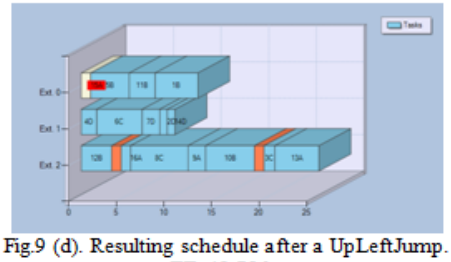

Fig.9 (d). Resulting schedule after a UpLeftJump. TT=42.78 h.

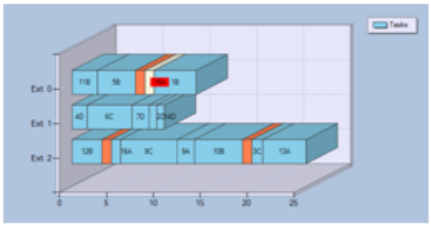

Fig.9 (e). Resulting schedule after a RightSwap. TT=39.49 h.

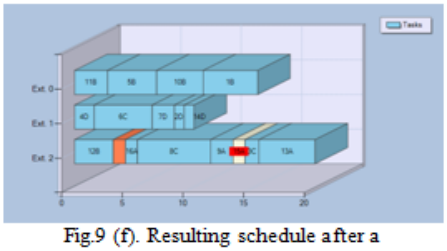

Fig.9 (f). Resulting schedule after a DownRightSwap. TT=29.60 h.

Fig. 9. Example of applying the optimal sequence of repair operators

The focal and global variables used in this example are detailed in [23], combined with a relational representation of the schedule that has been showed in previous section. To illustrate the advantages of RRL in real-time rescheduling, we consider the specific situation where there exist 14 orders already scheduled in the plant and a new order #15 must be inserted so that the Total Tardiness (TT) in the schedule is minimized. In each episode, a random schedule state for orders #1 through #14 is 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. The goal state for the repaired schedule is stated in terms of the TT: the order #15 must be inserted without increasing the TT present in the system.

Background knowledge such as “the number of orders scheduled for extruder #3 is larger than the number for extruder #2” is provided to speed up learning in the relational domain.

After 60 training episodes, only 7 repair steps are required, on average, to insert the 15th order. Fig. 13 provides an example of applying the optimal sequence of repair

operators from the schedule in Fig.13 (a). Before the 15th order has been included, the Total Tardiness is 37.60 h. Once the arriving order (in white) has been inserted, the Total Tardiness has been increased to 45.52 h; orange tasks are used to indicate cleaning operations. Based on the learned repair policy, several repair operators are applied, until the goal state is reached with a Total Tardiness of 29.60 h., which is even lower than the TT in the initial schedule before the 15th order was inserted.

6 Concluding Remarks

A novel approach for simulation-based development of a relational policy for automatic repair in real time of schedules using reinforcement learning and a prototype application, have been proposed. The policy allows generation of a sequence of deictic (local) repair operators to achieve rescheduling goals to handle abnormal and unplanned events, such as inserting an arriving order with minimum tardiness based on relational (deictic) representation of abstract schedule states, and repair operators. Representing schedule 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. These repair policies relies on abstract states, which are induced with general purposes, allowing the use of a compact representation of the problem which is are independent of the type of event that has been used to generate the disruption. 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, extruder 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 examples (triplets).
The main difference of the proposed approach with other reactive scheduling techniques is that rescheduling knowledge is generated based on plant-specific structure and events. Also, it can easily accommodate shop-floor decision rules to advantage. Moreover, in *SmartGantt* each response to a disruptive event is goal-oriented and aims to find a "satisfactory" solution whereas in existing literature on reactive scheduling the search for an optimal solution is pervasive.
Current work is focused on extending *SmartGantt* to an industrial case study with +100 products and several production tasks (formulation, testing, filling, etc.) with intermediate storage and waiting times. Schedule states for multi-stage configurations and WIP constraints are rather straightforward in RRL by resorting to the batches or orders as the focal points for deictic representations.

References

1. Vieira, G., Herrmann, J. Lin, E.: Rescheduling Manufacturing Systems: a Framework of Strategies, Policies and Methods. J. of Scheduling, 6, 39 (2003)

2. Méndez, C., Cerdá, J., Harjunkoski, I., Grossmann, I., Fahl, M.: State-of-the-art Review of Optimization Methods for Short-term Scheduling of Batch Processes. Computers and Chemical Engineering, 30, 913 (2006)
3. Henning, G., Cerda, J.: Knowledge-based Predictive and Reactive Scheduling in Industrial Environments. Computers and Chemical Engineering, 24, 2315 (2000)
4. Henning, G.: Production Scheduling in the Process Industries: Current Trends, Emerging Challenges and Opportunities. Computer-Aided Chemical Engineering, 27, 23 (2009)
5. Adhitya, A., Srinivasan, R., Karimi, I. A.: Heuristic Rescheduling of Crude Oil Operations to Manage Abnormal Supply Chain Events. AIChE J., 53, No. 2, p. 397 (2007)
6. Miyashita, K., Sycara, K.: CABINS: a Framework of Knowledge Acquisition and Iterative Revision for Schedule Improvement and Iterative Repair. Artificial Intelligence, 76, 377 (1994)
7. Zweben, M., Davis, E., Doun, B., Deale, M.: Iterative Repair of Scheduling and Rescheduling. IEEE. Trans. Syst. Man Cybern., 23, 1588 (1993)
8. Miyashita, K.: Learning Scheduling Control through Reinforcements, International. Transactions in Operational Research (Pergamon Press), 7, 125 (2000)
9. Croonenborghs, T.: Model-assisted Approaches to Relational Reinforcement Learning. Ph.D. dissertation, Department of Computer Science, K. U. Leuven, Leuven, Belgium (2009)
10. Džeroski, S., De Raedt, L., Driessens, K..: Relational Reinforcement Learning. Machine Learning, 43, No. 1/2, p. 7 (2001)
11. De Raedt, L.: Logical and Relational Learning. Springer-Verlag, Berlin (2008)
12. Sutton, R., Barto, A.: Reinforcement Learning: An Introduction. MIT Press, Boston, (1998)
13. Martinez, E.: Solving Batch Process Scheduling/Planning Tasks using Reinforcement Learning, Computers and Chemical Engineering, 23, S527 (1999)
14. Van Otterlo, M.: The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for Adaptive Sequential Decision Making Under Uncertainty in First-order and Relational Domains, IOS Press, Amsterdam (2009)
15. Blockeel, H., De Raedt, L.: Top-down Induction of First Order Logical Decision Trees. Artificial Intelligence, 101, No. 1/2, p. 285 (1998)
16. Driessens, K., Ramon, J., Blockeel, H.: Speeding up Relational Reinforcement Learning through the use of an Incremental First Order Decision Tree Learner. In: De Raedt, L. and Flach, P. (eds.) 13th European Conference on Machine Learning, vol. 2167, 97, Springer, Heidelberg (2001)
17. Driessens, K., Ramon, J.: Relational Instance Based Regression for Relational Reinforcement Learning. In: 20th International Conference on Machine Learning, 123, AAAI Press, Washington (2003)
18. Driessens, K., Ramon, J., Gärtner, T.: Graph Kernels and Gaussian Processes for Relational Reinforcement Learning. Machine Learning, 64, No. 1/3, 91 (2006)
19. Driessens, K., Džeroski, S.: Integrating Guidance into Relational Reinforcement Learning. Machine Learning, 57, 271 (2004)
20. Gärtner, T.: Kernels for Structured Data. Series in Machine Perception and Artificial Intelligence, Vol. 72, World Scientific Publishing, Singapore (2008)
21. Blockeel, H., De Raedt, L., Jacobs, N., Demoen, B.: Scaling up Inductive Logic Programming by Learning from Interpretations. Data Mining and Knowledge Discovery vol. 3(1) pp. 59-93 (1999)
22. Musier, R., Evans, L.: An Approximate Method for the Production Scheduling of Industrial Batch Processes with Parallel Units. Computers and Chemical Engineering, 13, 229 (1989)
23. Palombarini, J., Martínez, E.: Learning to Repair Plans and Schedules using a Relational (deictic) Representation. Brazilian Journal of Chemical Engineering vol. 27, 03, pp. 413-427 (2010)