Automated Task Rescheduling using Relational Markov Decision Processes with Logical State Abstractions

Jorge Palombarini[1], Ernesto Martinez[2]

[1] GISIQ(UTN) - Fac. Reg. Villa 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. Generating and representing knowledge about heuristics for repair-based scheduling is a key issue in any rescheduling strategy to deal with unforeseen events and disturbances. Resorting to a feature-based representation of schedule states is very inefficient and generalization to unseen states is highly unreliable whereas the acquired knowledge is difficult to transfer to similar scheduling domains. In contrast, first-order relational representations enable the exploitation of the existence of domain objects and relations over these objects, and enable the use of quantification over objectives (goals), action effects and properties of states. In this work, a novel approach which formalizes the rescheduling problem as a Relational Markov Decision Process integrating first-order (deictic) representations of (abstract) schedule states is presented. The proposed approach is implemented in a real-time rescheduling prototype, allowing an interactive scheduling strategy that may handle different repair goals and disruption scenarios. The industrial case study vividly shows how relational abstractions provide compact repair policies with less computational efforts.

Keywords. Rescheduling, Relational Markov Decision Process, Manufacturing Systems, Reinforcement Learning, Abstract States.

1 Introduction

In the context of manufacturing systems established production planning and control systems must cope with unplanned events and intrinsic variability in dynamic environments where difficult-to-predict circumstances occur as soon as plans are released to the shop-floor [1]. 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, which is a complex phenomenon that cannot be addressed exclusively through the inclusion of uncertain parameters into problem statement [2]. 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 [3]. 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, producing satisfactory schedules rather than optimal ones in reasonable computational time.

Exploiting peculiarities of the specific problem structure is the main aim of the vast majority of the scheduling research prioritizing schedule efficiency using a mathematical programming approach, in which the repairing logic is not clear to the end-user [4,5,6]. More recently, Li and Ierapetritou [7] have incorporated uncertainty in the form of a multi-parametric programming approach for generating rescheduling knowledge for specific events. Also, Gersmann and Hammer [8] have developed an improvement over an iterative schedule repair strategy using Support Vector Machines. However, the tricky issue is that resorting to a feature-based representation of the schedule state is very inefficient, generalization to unseen states is risky and knowledge transfer to unseen scheduling domains is not feasible [9]. Therefore, at the representation level, it is mandatory to scale up towards a richer formalism that allows the incorporation of learning/reasoning skills [10]. In that sense, Relational Markov Decision Processes are a natural choice because they can be solved by means of simulating state transitions and enabling the integration of first-order relational representations. Thereby, exploitation of the existence of domain objects and relations (or properties) over these objects, quantification over objectives (goals), action effects and properties of schedule states, abstraction and generalization processes can be carried on in a straightforward way.

In this work, a novel real-time rescheduling approach which resorts to a Relational Markov Decision Process to integrate (deictic) representations of (abstract) schedule states with repair operators is presented. To learn a near-optimal policy for rescheduling using simulations, 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 [11]

2 Relational Markov Decision Processes

Relational Markov Decision Processes (RMDP), are an extension from standard MDPs based on relational representations in which states correspond to Herbrand interpretations [10], and can be defined formally as follows [12]:

Definition 1. Let $P = \{p_1/\alpha_1, \ldots, p_n/\alpha_n\}$ be a set of first order predicates with their arities, $C = \{c1, \ldots, ck\}$ 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 rescheduling problem as a RMDP enables 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). An RMDP can be solved using a Relational Reinforcement Learning (RRL) algorithm, where schedule states are represented as sets of first-order logical facts, and the learning algorithm can only see one state at a time. Repair operators are also represented relationally as predicates describing each feasible action in a given schedule state as a relationship between one or more variables, as it is shown in Example 1 below. 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 [11,12,13]. 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.

Example1 .

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

```
ALGORITHM 1: SCHEDULE STATE TRANSITION
1. Examples ← Ø
2. BK ← Background Knowledge Prolog Rules
3. Repeat
4.   Initialize the Q-function hypothesis
5.   Induce Abstract State Membership Function
     ψ using Examples, BK and TILDE Algorithm
6.   e ← 0
7.   Repeat
8.     Generate a starting schedule state ψ(s_0)
9.     i ← 0
10.    Repeat
11.      choose a repair operator a_i at s_i using a
         policy (e.g., ε-greedy) based on the
         current hypothesis
12.      implement operator a_i, observe r_i
         and the resulting schedule ψ(s_i+1)
13.      i ← i+1
14.    Until schedule state ψ(s_i) is a goal state
15.    For j←i-1 to 0 do
16.        generate example x ← (ψ(s_j), a_j, Q_j), where
           Q_j ← r_j + γ max_a Q(ψ(s_j+1),a)
17.        Examples ← Examples ∪ {x}
18.    Next
19.    Update Q_e using Examples and a relational
         regression algorithm (e.g. TG)
20.    e ← e+1
21. Until no more learning episodes
22.Until no more trials
```

```
ALGORITHM 2: STATE ABSTRACTION
Require
s ← actual schedule ground state in
relational format.
P ← collection of prolog rules
induced by TILDE, which represents
the Abs. State Membership Function ψ
BK ← collection of background know-
ledge Prolog rules
ABS ← Ø

1. For each rule available in P
2.   AbstractState ← body(rule)
3.   ABS.Add(AbstractState)
4. Next
5. Consult BK, ABS
6. For each absState available in ABS
7.   if absState θ-subsumes s then
8.     return absState
9.   end if
10.Next
11.return Ø
```

Fig. 1. Adapted RRL algorithm for learning to repair schedules through schedule state transition simulation (Algorithm 1) and state abstraction (Algorithm 2).

Rather than using an explicit state-action Q-table, RRL stores the Q-values in a logical regression tree [14]. The relational version of the Q-learning algorithm is shown in Fig. 1 (Algorithm 1). So, BK is loaded before the training process starts and for each trial after the Q-function hypothesis has been initialized, the RRL algorithm starts

running learning episodes [11, 15]. During each episode, all the visited abstract states and the selected actions are stored, together with the rewards related to each visited (*abstract state*, *action*)-pair. At the end of each episode, when the system ends up in 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 (*abstract state*, *action*)-pair in the episode. The algorithm presents the set of (*abstract state*, *action*, *qvalue*)-triplets encountered in each learning episode to a relational regression engine, which will use this set of *Examples* to update the current regression tree for the *Q*-function. In order to accelerate and generalize the learning process an abstract state induction is performed based on ground schedule states. 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* (BK) which is generally true for the entire domain to facilitate induction of abstract states and repair policy in the form of Prolog rules. 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.

TILDE relational regression algorithm [13,15,16] is used by the prototype to induce the membership function ψ which allows obtaining the abstract-state corresponding to the ground state. Such function is used by Algorithm 2 in Fig. 1 which carries on this task. TG algorithm is used for accumulating simulated experience in a compact way at the end of each learning episode, in a yet readily available decision-making rule for generating a sequence of repair operators that are available at each abstract schedule state *s*

2.1 Retrieving experience and membership values using logical decision trees

All accumulated experience and the membership function ψ are 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. 2). Also, every leaf (terminal node) of the tree involves a real valued prediction. Prediction with first-order trees is similar to prediction with propositional decision trees: every new instance is sorted down the tree.

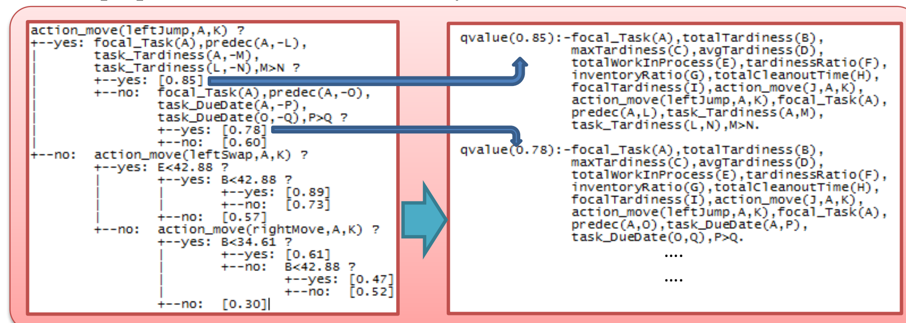


Fig. 2. An example of a part of a relational regression tree (left) and a detail of two from ten possible derived Prolog rules (right), without root query.

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 the prototype in a set of Prolog rules that are used in execution time to predict *Q*-values and to 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 since it is the first node of the tree upon which further refinements will be performed in the "Update $\hat{q}$" line of Algorithm 1. Furthermore, the Root Query generates a set of basic variables used by the TG to execute several tests that are needed 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 heuristics. Theses possible tests are taken from the background knowledge definition, 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 to 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) could be ← precedes(X,Y), orderOfProduct(X,Product), and ← precedes(X,Y), task_in_resource(X,Resource), in which orderOfProduct/2 and task_in_resource/2 can be defined in the set of facts that define the schedule state or might be derived using BK rules. Several heuristic functions can be used to determine the best tests, and to decide when to turn nodes into leaves. The function employed by the TG algorithm is based on the information gain, which measures the amount of information gained by performing a particular test [13].

3 Abstraction and Generalization in Rescheduling Domains

The drawbacks and limitations of attribute-value (propositional) representations in learning a rescheduling policy discussed in previous sections are solved by our proposal 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 repair 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 P and a set of constants C. Each constant c ∈ C denotes an object (i.e. a task or resource) in the domain and each p/a ∈ P denotes either a property (or attribute, i.e. task tardiness) of some object (if a=1) or a relation between objects (for example, if a > 1, e.g. precedes(task1,task2)).

To represent *structured terms* in the schedule domain, e.g., resource(1,extruder(1),[task1,task2,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. The prototype implements the concept of "learning from interpretations" in [13], 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 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 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 and due date in a rush order at the top of the priority list.

3.1 Abstract Schedule State Spaces

Relational representations of schedule states and repair operators are *symbolic* in nature. Therefore, to perform the process of generalizing and abstracting over them to speed up learning, the TG algorithm is used in combination with the Algorithm 2 shown in Fig. 1 which is based in the concept of first-order abstractions of version spaces. This feature has been implemented following the Policy Iteration using Abstraction and Generalization Techniques (PIAGeT) principle [12], whose instantiation is given in Fig. 3. The base level of the figure consists of the original, flat RMDP. The *abstraction level* defines a representation of this RMDP that abstracts parts of the learning process, value functions and repair policy. The relational representation can be viewed as a particular homomorphism of the flat model. This abstraction level is given a priori as a particular bias, based on background knowledge and assumptions and restrictions about the domain (e.g. BK also might specify constraints such as BK: ∀XY(precedes(X,Y) → X /= Y), saying that if task X precedes task Y, they should be different objects). Relational ground states are aggregated into abstract states using

TILDE, which uses an inductive logic programming approach for generating logical regression trees in an off-line way.

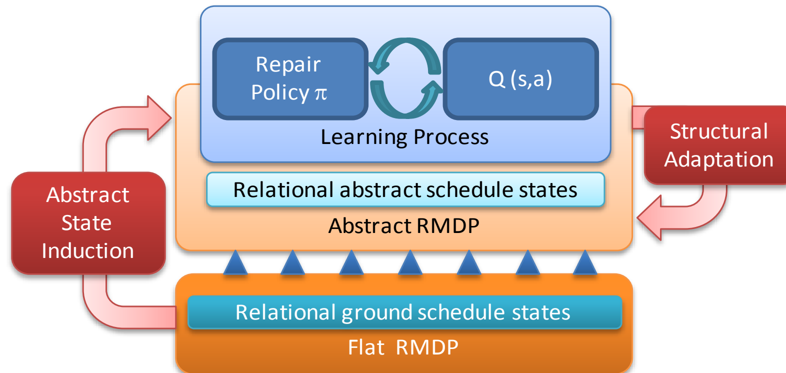


Fig 3. Abstraction and Generalization in rescheduling domains using PIAGeT principle [12].

Q-values can be learned for all abstract states and the derived abstract policy maps these abstract schedule states into repair actions aiming at achieving a given repair goal or objective. There are two interacting processes in our proposal: the base layer is the *structural* part for which an abstract layer is defined and learned, and the top layer that is responsible for learning a repair policy. Finally, the Structural Adaptation component is added so as to the defined structures are allowed to change as the learning process incorporates more training examples via simulation or through knowledge provided by experts in the form of BK rules.

Example 3. $\forall A,B,C,D$(maxTard(A), precedes(B,C), precedes(C,D), A>57.8) is an Abstract State that denotes the set of schedules states in which the maximum 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.

As can be seen in Example 3, each abstract state models a set of interpretations of the underlying learning process (RMDP), and defines which relations should hold in each of the ground states it covers. Formally, this is expressed as a conjunction $\equiv 1\wedge . . . \wedge m$ of logical atoms, e.g., a logical query. The use of variables makes room for abstracting over specific domain objects as well. Thus, an abstract state is basically a logical sentence, specifying general properties of several states visited during learning through simulation of abstract state transitions. The action-value Q-function relies on a set of abstract states, which together encode the kind of rescheduling knowledge learned through state transition simulation in a compact and formal way, which can be used in real time to repair plans whose feasibility have been affected by disruptive events. 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 [12]:

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 $\lfloor \varphi_1, \ldots, \varphi_n \rceil$, where each φ_i (i=1 . . . n) (called a **part**) is a formula. A structure $\mathcal{A} \in \mathbb{A}$ is covered by an MPA iff there exists a part φ_i (i=1 . . . n) such that $\mathcal{A} \models \varphi_i$. 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 $\sigma \models \varphi$. Then, an **abstract schedule state space** is an MPA $\lfloor \varphi_1, \ldots, \varphi_n \rceil$, where each φ_i (i=1…n) is an abstract schedule state. An **abstract schedule state action space** is an MPA $\lfloor \langle \varphi_1, \alpha_1 \rangle \ldots, \langle \varphi_n, \alpha_n \rangle \rceil$, i.e. a product-MPA over the schedule state-action space $S \times A$.

Definition 3. Let $M = \langle S, A, T, R \rangle$ be an RMDP. An abstract RMDP $\mathbf{M}$ is a structure $\langle Z, A, T, R \rangle$. The abstract schedule state space $Z = \{Z_1, . . . , Z_n\}$ is a partition of the schedule state space S. Typically, $|S| >> |Z|$, offering a solution to many of the problems with large state spaces described previously. Let ψ denote the membership function defined as $\psi : S \rightarrow Z$, which maps each schedule state in the original space S to one of the sets in Z. Now each partition in Z is defined as $Z_i = \{s \,/\, \psi(s) = Z_i\}$. For all i, $j = 1. . . n$, Z satisfies the following properties: i) $Z_i \subseteq S$, ii) $\bigcup_i Z_i = S$, and iii) $Z_i \cap Z_j = \emptyset$, if i = j. Since we do not consider action-space abstraction, both M and $\mathbf{M}$ share the same action set A. A transition function and a reward function for $\mathbf{M}$ can now be defined in terms of T and R.

$$\mathrm{R}(\mathrm{Z_i}, \mathrm{a}) = \sum_{\mathrm{s} \epsilon \mathrm{Z_i}} \omega(\mathrm{s}) \cdot R(\mathrm{s}, \mathrm{a}) \tag{1}$$

$$\mathrm{T}(\mathrm{Z_i}, \mathrm{a}, \mathrm{Z_j}) = \sum_{\mathrm{s} \epsilon \mathrm{Z_i}} \sum_{\mathrm{s'} \epsilon \mathrm{Z_j}} \omega(\mathrm{s}) \cdot T(\mathrm{s}, \mathrm{a}, \mathrm{s'}) \tag{2}$$

To ensure that T and R are well-defined, a weighting function $\omega : S \rightarrow [0,1]$ has been added, where for each $Z_i \epsilon Z, \sum_{s \epsilon Z} \omega(s) = 1$. The weighting $\omega(s)$ expresses how much the state s contributes to the abstract state $Z_I = \psi(s)$. The function ω is chosen to be in proportion to the state occupancy distribution using the TG algorithm online. Therefore, we learn an abstract policy Π which is defined as $\Pi : Z \rightarrow A$ such that $\pi(s, a) = \Pi(\psi(s), a)$, for all $s \in S$, $a \in A$. So, in this approach it is necessary to induce an abstract schedule state space Z for which the Q-values and policies are learned. Q-learning on an abstract schedule state space is performed by updating an abstract action-value function based on the (ground) transition (s_t, a_t, r_t, s_{t+1}) as follows:

$$Q(\psi(s_t), \mathrm{a_t}) := Q(\psi(s_t), \mathrm{a_t}) + \alpha(\mathrm{r_t} + \gamma \cdot \max_{\mathrm{a'}} Q(\psi(s_{t+1}), \mathrm{a'}) - Q(\psi(s_t), \mathrm{a_t})) \tag{3}$$

In this way, Q-values for abstract states are learned, and these Q-values are shared among all states that are members of the same abstract state (See Algorithm 1 in Fig. 1). Based on the presented RRL approach, the prototype 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 schedules states. These

partitions are described by a kind of *abstract* schedule state, that is, a logical condition, which matches several real schedule states like the one in Example 3. 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 depicted in Fig. 1 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, which is performed using ψ. That is, the abstract state S 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 corresponding repair policy expresses takes advantage of the problem structure and relations among objects in the schedule domain [9]. As a result, the rescheduling policy may be reused in somewhat similar problems were the same relations apply, and without any further learning. As only fully observable worlds are considered, rescheduling knowledge is only due to abstracting schedules using relationships between world objects. An abstract state S covers a ground state s iff $s \models S$, which is tested using θ-subsumption.

4 Industrial Case Study

An example problem proposed in [17] is considered to illustrate our approach for automated task rescheduling. It consists of a batch plant which is made up of 3 semi-continuous extruders that process customer orders for four products (A,B,C and D). Each extruder has distinctive characteristics, 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 [17].

The prototype application has been implemented in Visual Basic.NET 2008 Development Framework 3.5 SP2 and SWI Prolog 6.0.2 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 [16] have been used. The prototype admits two modes of use: *training* and *consult*. During training it 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 mode. 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. Logical queries are 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 ob-

jects describing schedule states. Before starting a training session the user must define through a graphical interface, the value of all simulation and training parameters, related to *Initial schedule condition, Learning Parameters* and *Goal State Definition*. The latter is a key parameter that has to be defined before starting simulation since it establishes the desired repair goal. Training a rescheduling agent can be carried out by selecting one of three alternative goals, which is selected through an option list: *Tardiness Improvement*, *Stability or Balancing*. 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 corresponding one for the initial schedule.

Order attributes correspond to product type, due date and size. In learning to insert an order, the rescheduling scenario is described by: i) arrival of an order with given attributes that should be inserted into 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 scenarios 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 (randomly between [10,20]), Order composition, Order Size (an interval between 100 y 1000 kg) and Due Date.

The focal and global variables used in this example are shown in Example 1, in all cases training is carried out with a variable number of orders in a range of 10 to 15. 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. In each training episode, a random schedule state was generated, and a random insertion is attempted for the new order (whose attributes are also randomly chosen), which in turn serves as the focal point for defining repair operators. Fig. 4 higlights results in 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.

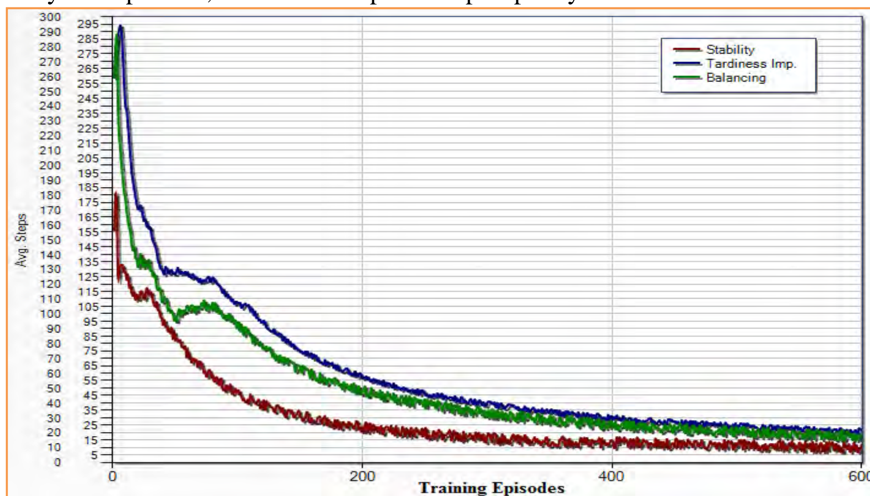
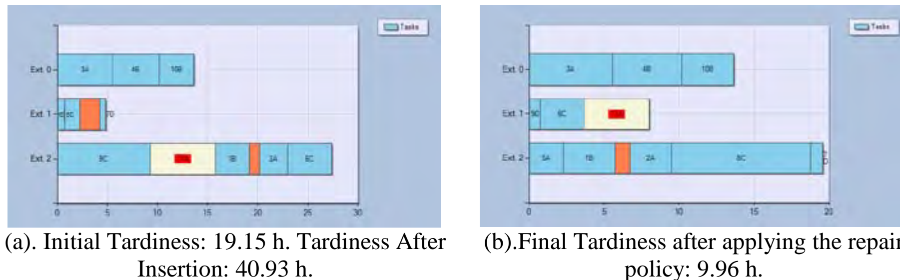


Fig.4. Learning curve for the Arrival of a new order event, with three different goals.

For the other two more stringent situations, namely Tardiness Improvement and Balancing, learning curves tend to stabilize later, possibly due to a higher number of repair operations which are necessary to try at early stages of learning so as to guarantee goal achievement. As it is shown in Fig.4, 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. 5 provides an example of applying the optimal sequence of repair operators from the schedule in Fig.5 (a), using the Consult mode of the prototype, 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 randomly 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 sequence *RightJump-UpLeftSwap-LeftMove-DownRightSwap-LeftSwap-upRightSwap* is applied until 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.



(a). Initial Tardiness: 19.15 h. Tardiness After Insertion: 40.93 h.

(b).Final Tardiness after applying the repair policy: 9.96 h.

Fig.5. Repairing sequence for Arrival of a new order event

As can be seen in the generated repair sequence, the rescheduling 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 to handle order insertion. As it is shown in Fig.5, 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 still remains.

5 Concluding Remarks

A novel approach for simulation-based learning of a relational policy dealing with automated repair in real time of schedules based on Relational Markov Decision Processes has been presented. The policy allows generating 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 using 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 focal points and objects for defining repair strategies are typically used. These repair policies rely 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 states and macro-actions for schedule repair facilitate and accelerates learning and knowledge transfer, which is independent of the type of event that has generated a disruption and can be used reactively in real-time. Finally, 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 an end-user.

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. Aytug, H., Lawley, M., McKay, K., Mohan, S., Uzsoy, R.: Executing production schedules in the face of uncertainties: A review and some future directions. European Journal of Operational Research, 161, 86–110 (2005)
3. Henning, G.: Production Scheduling in the Process Industries: Current Trends, Emerging Challenges and Opportunities. Computer-Aided Chemical Engineering, 27, 23 (2009)
4. Adhitya, A., Srinivasan, R., Karimi, I. A.: Heuristic rescheduling of crude oil operations to manage abnormal supply chain events. AIChE J. 53(2), 397-422 (2007)
5. Miyashita, K.: Learning Scheduling Control through Reinforcements, International. Transactions in Operational Research (Pergamon Press), 7, 125 (2000)
6. Zhu, G., Bard, J., Yu, G.: Disruption management for resource-constrained project scheduling. Journal of the Operational Research Society, 56, 365-381 (2005)
7. Li, Z., Ierapetritou, M.: Reactive scheduling using parametric programming. AIChE J. 54(10), 2610-2623 (2008)
8. Gersmann, K., Hammer, B.: Improving iterative repair strategies for scheduling with the SVM. Neurocomputing, 63, 271–292 (2005)
9. Morales, E. F.: Relational state abstraction for reinforcement learning. Proceedings of the Twenty-first Intl. Conference (ICML 2004), Banff, Alberta, Canada, July 4-8 (2004)
10. Palombarini, J., Martínez, E.: SmartGantt – An Intelligent System for Real Time Rescheduling Based on Relational Reinforcement Learning. Expert Systems with Applications vol. 39, pp. 10251- 10268 (2012)
11. Džeroski, S., De Raedt, L., Driessens, K..: Relational Reinforcement Learning. Machine Learning, 43, No. 1/2, p. 7 (2001)
12. 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)
13. De Raedt, L.: Logical and Relational Learning. Springer-Verlag, Berlin (2008)
14. Blockeel, H., De Raedt, L.: Top-down Induction of First Order Logical Decision Trees. Artificial Intelligence, 101, No. 1/2, p. 285 (1998)
15. Sutton, R., Barto, A.: Reinforcement Learning: An Introduction. MIT Press (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 Euro Conf. Machine Learning, vol. 2167, 97, Springer, (2001)
17. Musier, R., Evans, L.: An Approximate Method for the Production Scheduling of Industrial Batch Processes with Parallel Units. Comp. and Chem. Engineering, 13, 229 (1989)