Enriching a Temporal Planner with Resources and a Hierarchy-based Heuristic

Alessandro Umbrico (✉)[1], Andrea Orlandini[2], Marta Cialdea Mayer[1]

[1] Dipartimento di Ingegneria
Università degli Studi Roma Tre
[2] Istituto di Scienze e Tecnologie della Cognizione
Consiglio Nazionale delle Ricerche, Roma

Abstract. A key enabling feature to deploy a plan-based application for solving real world problems is the capability to integrate Planning and Scheduling (P&S) in the solving approach. *Flexible Timeline-based Planning* has been successfully applied in several real contexts to solve P&S problems. In this regard, we developed the *Extensible Planning and Scheduling Library* (EPSL) aiming at supporting the design of P&S applications. This paper describes some recent advancements in extending the EPSL framework by introducing the capability to reason about different types of "components", i.e., *state variables* and *renewable resources*, and allowing a tight integration of Planning and Scheduling techniques. Moreover, we present a domain independent *heuristic* function supporting the solving process by exploiting the hierarchical structure of the set of timelines making up the flexible plan. Some empirical results are reported to show the feasibility of deploying an EPSL-based P&S application in a real-world manufacturing case study.

1 Introduction

The Timeline-based planning approach has been successfully applied in several real world scenarios, especially in space like contexts [1,2,3]. Besides these applications, several timeline-based Planning and Scheduling (P&S) systems have been deployed to define domain specific applications, see for example EUROPA [4], IXTET [5], APSI-TRF [6]. However, despite their practical success, these systems usually entails the development of applications closely connected to the specific domain they are made for. As a consequence, it is not straightforward to adapt these applications to domains requiring different solving capabilities and thus, it is often necessary to define new solvers somehow loosing "past experiences". To address the above issue, a research initiative has been started to develop a domain independent *Extensible Planning and Scheduling Library* (EPSL) [7]. EPSL aims at defining a modular and extensible software environment to support the development of timeline-based applications. The structure of EPSL allows to preserve "past experiences" by providing a set of ready-to-use algorithms, strategies and heuristics that can be combined together. In this way, it is possible to develop/evaluate several solving *configurations* in order to find the one which best fits the features of the particular domain to be addressed.

In this paper two enhancements of the EPSL framework are presented. First, the possibility to model and manage renewable resources is introduced in the framework. Briefly, a renewable resource is a shared component having a limited capacity, that is however not consumed by the processes using it: when it is released, it returns to its full capacity. Examples of renewable resources are the memory of a software device or a machine that can process a limited number of pieces at a time. Renewable resources allow the planning framework to model more realistic domains. Secondly, a domain independent heuristic function is defined exploiting the structure of the timelines and the dependencies among them induced by the rules constraining the domain. Such a structure conveys important information that can be used to improve the performances of the planner. It is worth pointing out that domains modeled following a hierarchical approach often exhibit this kind of structure of the system components. An experimental evaluation on problem domains derived from a real-world manufacturing context are finally presented to assess the deployment of the above mentioned heuristic.

2 Timeline-based planning in a nutshell

Timeline-based planning has been introduced in early 90s [1] and several formalizations have been proposed for this approach [8,9,10]. The timeline-based approach takes inspiration from control theory. It models a complex domain by identifying a set of relevant features that must be controlled over time. Domain features are modeled by means of state variables, a set of rules that "locally" constrains the temporal evolutions of related features and describing the allowed sequence of values/states a feature can assume over time. Domain features are further constrained by means of *synchronization rules*, i.e. a set of "global" constraints that allow one to coordinate different features in order to obtain consistent behaviors of the overall system. A timeline-based planner uses these rules (called *domain theory*) to build *timelines* for domain features. A timeline is a sequence of valued temporal intervals, called *tokens*, each of which specifies the value assumed by the state variable in that interval. So, a timeline describes the temporal evolution/behavior of the related feature over time. The start and end points of the tokens making up a *flexible timeline* are temporal intervals instead of exact time points [8]. A flexible timeline represents an envelope of possible evolutions of the associated feature that can be exploited by an executive system for a robust online execution of the plan [11,12]. *Flexible Timeline-based Planning* usually follows a partial order planning approach starting from a set of partially defined timelines (i.e. the initial planning problem) and building (if possible) a set of completely instantiated timelines (i.e. a solution plan) within a given temporal horizon.

2.1 A Hierarchical Modeling Approach

This section is devoted to briefly present a general methodology, often used when modeling a domain in the timeline-based style. This approach generates a

structure in the model which can be exploited during the solving process. The essential of the methodology is a decomposition analysis (like in [13,14]), aiming at identifying the "relevant" features (system's components) that independently evolve over time. A generic component is then described by a set of activities to carry out and the logical states the system can assume coupled with related timing and causal constraints, i.e. temporal durations as well as allowed state transitions. This approach results in a hierarchical model of the domain, with higher level components abstracting away from the internal structure of the system to be controlled. While deepening the analysis into details, the concrete features of the system are represented.

The modeling approach described here usually identifies three relevant classes of components [15]: (i) *functional*, (ii) *primitive* and (iii) *external* components. A *functional* component provides a logical view of the system as a whole in terms of what the system can do notwithstanding its internal composition. It models the high-level functionalities the system is able to perform. A *primitive* component provides a logical view of a particular element composing the system. Usually, values of such a component correspond to concrete states/actions the related element is able to assume/execute in the environment. Finally, an *external* component provides a logical view of elements whose behaviors are not under the control of the system but affect the execution of its functionalities. They model conditions that must hold in order to successfully perform internal activities. In addition to the description of single state variables, their behaviors are to be further constrained by specifying inter-components causal and temporal requirements (called *synchronization rules* in the timeline-based approach) allowing the system to coordinate its sub-elements while safely realizing complex tasks. In this regard, following a hierarchical approach, such rules map the high-level functionalities of the system into a set of activities on primitive and/or external components enforcing operational constraints that guarantee the proper functioning of the overall system and its elements. Namely, synchronizations allow to specify how the high-level functionalities, modeled by means of functional components, are related to the primitive and external components of the domain. The synchronization rules of the domain often reflect the hierarchy of the system components: the values of a higher level (more abstract) state variable are constrained to occur while suitable values are assumed by corresponding lower level ones, modeling its more concrete counterparts.

3 The Extensible Planning and Scheduling Library

EPSL is a layered framework built on top of APSI-TRF[1], it aims at defining a flexible software environment for supporting the design and development of timeline-based applications. The key point of EPSL flexibility is its interpretation of a planner as a "modular" solver which combines together several elements

[1] APSI-TRF is a software framework developed for the European Space Agency by the Planning and Scheduling Technology Laboratory at CNR (in Rome, Italy) for supporting the design and deployment of timeline-based P&S applications.

to carry out its solving process. The main components of the EPSL architecture are the following. The *Modeling layer* provides EPSL with timeline-based representation capabilities. It allows to model a planning domain in terms of timelines, state variables, synchronizations and to represent flexible plans. The *Microkernel layer* is the key element which provides the framework with the needed flexibility to "dynamically" extend the framework with new elements. It is responsible to manage the lifecycle of the solving process and the elements composing the application instances (i.e. the planners). The *Search layer* and the *Heuristics layer* are the elements responsible for managing strategies and heuristics a planner can use during the solving process to support the search. The *Engine layer* is the element responsible for managing the portfolio of algorithms, called resolvers, available. Resolvers characterize the expressiveness of EPSL-defined planners. Namely they define what a planner can actually do to solve problems. Finally the *Application layer* is the top-most element which carries out the solving process and finds a solution if any.

3.1 The Epsl Solving Procedure

The solving procedure of a generic EPSL-based planner is described in Algorithm 1. It consists in a plan refinement procedure which iteratively *refine*s a plan π by detecting and solving conditions, called *flaws*, that affect the completeness and/or consistency of π. EPSL instantiates the planner solving process over the tuple $\langle \mathcal{P}, \mathcal{S}, \mathcal{H}, \mathcal{E} \rangle$ where $\mathcal{P}$ is the specification of a timeline-based problem to solve, $\mathcal{S}$ is the search strategy the planner uses to expand the search space, $\mathcal{H}$ is the heuristic function the planner uses to select the most promising flaw to solve, and $\mathcal{E}$ is a set of resolvers the planner uses to detect flaws of the plan and compute their solutions.

Algorithm 1 solve$(\mathcal{P}, \mathcal{S}, \mathcal{H}, \mathcal{E})$

```
1: // initialize search
2: π ← InitialPlan(P)
3: fringe ← ∅
4: // check if plan is complete and flaw-free
5: while ¬IsSolution(π) do
6:     Φ ← DetectFlaws(π, E)
7:     // check the set of flaws
8:     if Φ ≠ ∅ then
9:         // select the most promising flaw to solve
10:        φ ← SelectFlaw(Φ, H)
11:        // call resolver to detect flaws and compute solutions
12:        for resv ∈ E do
13:            nodes ← HandleFlaw(φ, resv)
14:            // expand the search with possible plan refinements
15:            fringe ← Enqueue(nodes, S)
16:        // check fringe
17:        if fringe = ∅ then
18:            // unsolvable flaws
19:            return Failure
20:    // go on with search - backtracking point
21:    π ← GetPlan(Dequeue(fringe))
22: // get solution plan
23: return π
```

The plan π is initialized on the problem description $\mathcal{P}$ (row 2) and then the procedure iteratively refines the plan until a solution or a failure is detected (rows 5-21). Plan refinement consists in detecting flaws and compute their solutions by means of resolvers $\mathcal{E}$ (rows 6-15). Given a set of flaws Φ of the plan, the most promising flaw ϕ to solve is selected according to heuristic $\mathcal{H}$ (row 10) and then resolvers compute possible solutions. Each solution represents a possible refinement of the current plan so a new branch of the search space is created for each of them and the resulting node is added to the fringe according to $\mathcal{S}$ (row 15). The search goes on until a plan with no flaws is found, i.e. a solution plan (row 5). However if the fringe is empty (rows 17-19) this means that there are unsolvable flaws in the plan, then the procedure returns a failure.

Algorithm 1 depicts a standard search procedure. It is important to point out that the particular set of resolvers $\mathcal{E}$, the strategy $\mathcal{S}$ and the heuristic $\mathcal{H}$ used can strongly affect the behavior and the performance of the solving process. Resolvers are the architectural elements allowing a planner instance to actually build the plan. A resolver encapsulates the logic for detecting and solving specific type of flaws. The greater is the number of available resolvers the greater is the number of flaw types an EPSL-based planner can handle. The set of available resolvers determine the "expressiveness" of the framework, i.e. the type of problems EPSL can solve. Broadly speaking a flaw represents a particular condition to solve for building a consistent and complete plan. It is possible to identify two main classes of flaws: (i) *goals*, flaws affecting the completion of a plan; (ii) *threats*, flaws affecting the consistency of a plan.

At any iteration of Algorithm 1 the refinement procedure detects the flaws of the current plan (row 6) and selects the most "promising" flaw to solve according to the heuristic $\mathcal{H}$ (row 10). Flaw selection is not a backtracking point of the search but it can strongly affect the performance of the solving procedure. A "good" choice of the next flaw to solve, indeed, can prune the search space by cutting off branches that would not bring to solutions. The heuristic $\mathcal{H}$ encapsulates the policy used by the planner to analyze and select plan flaws. In this regards EPSL framework allows an application designer either to develop a heuristic exploiting some domain-specific knowledge of the domain or develop a domain-independent heuristic to address different domains.

These elements represent the main flexibility features of the framework. Here is where users can focus their development efforts in order to customize EPSL-based applications for the particular problem to address. As a matter of fact EPSL architecture allows to easily extend the solving capabilities of the framework by integrating new implementations of the elements described above.

3.2 Integrating Resources

One of the contribution of the paper is the introduction of renewable resources to extend EPSL modeling capabilities. A renewable resource is a shared element of the domain needed during the execution of an action but is not consumed. The resource maximum capacity $\mathcal{C}$ limits the number of activities that can concurrently require the resource. The access to the resource must be properly man-

aged in order to satisfy its capacity constraint. Namely the amount of resource requirements of concurrent activities must not exceed the resource capacity $\mathcal{C}$. EPSL has been extended with the introduction of a new component type for modeling renewable resources. A component of this type specifies the capacity $\mathcal{C}$ of the resource, and requirement activities are temporally qualified by means of tokens specifying the used amount of the resource. The consumption profile of the resource is constituted by the set of tokens making use of it. Obvioulsy, the total amount of resource used by overlapping tokens must not exceed its maximal capacity.

The EPSL framework has been extended with the introduction of a dedicated resolver for detecting and solving flaws concerning the management of renewable resources. The resolver must schedule requirement tokens by detecting and solving peaks on the resource consumption profile. A peak is detected every time the total amount of resource required by a set of (temporally) overlapping tokens is higher then the capacity $\mathcal{C}$. The peak is resolved by posting precedence constraints between particular subsets of activities, called *Minimal Critical Set*s (MCSs). A subset of activities of a peak is an MCS if a precedence constraint between any pair of these activities removes the peak. The resolver has been implemented by adapting to timelines the algorithms described in [16].

3.3 Integrating Heuristics

The flaw selection heuristic $\mathcal{H}$ is the element supporting the flaw selection step (row 10) of the plan refinement procedure. At any iteration of Algorithm 1 the planner must select the next flaw to refine the plan. If no heuristic is given, all the flaws detected are "equivalent" and the planner can only make a random choice. Namely there is no information characterizing the importance of the flaws.

Often, however, the flaws of a plan can have dependencies: solving a flaw can solve or simplify the solution of other related flaws. Therefore it is important to make "good" choices in order to reduce the number of refinement steps needed to build the plan. Bad choices, indeed, may bring to an inefficient solving procedure. A flaw selection heuristic $\mathcal{H}$ provides a criteria the planner can use to identify the most relevant to handle.

The *Hierarchical Flaw-selection Heuristic* (HFH) is an evaluation criteria relying on the hierarchical structure of timeline-based plans. Timelines may be related one to the other by synchronization rules. Given two timelines A and B, a synchronization $S_{A,B}$ from timeline A to timeline B, typically implies a dependency of B from A. Namely, the presence of some token on the timeline B is due to the need of synchronizing with some other token on the timeline A. Therefore, analyzing synchronization rules it is possible to build a *dependency graph* (DG) of the timelines. Figure 1 shows a set of timelines with synchronization rules (the arrows connecting the timelines) and the resulting DG where nodes are timelines and edges are synchronization rules. A hierarchy describing relationships among timelines can be extracted from the obtained DG. An edge from a node A to a node B in the DG represents a dependency between the two corresponding timelines. Thus the hierarchical level of the timeline A is not

lower than the hierarchical level of B; if moreover no path connects B to A, A is at a higher level in the hyerarchy, i.e. it is more independent than B. If a timeline A depends from B and vice-versa (i.e. A and B are contained in a looping path in the DG), then A and B have the same hierarchical level, and they are said to be hierarchically equivalent. In general, if the DG has a root, i.e. a node with only outgoing edeges, it represents the most independent timeline of the hierarchy. For instance, the hierarchy extracted from the DG in Figure 1 is $A \prec B \prec C$ and $B \prec D$, while C and D are hierarchically equivalent, so A is the most independent timeline while C and D are the less independent ones.

Usually, the DG resulting from a planning domain built by applying the modeling approach described in section 2.1 generates a non-flat hierarchy (sometimes even an acyclic graph) that can be successfully exploited by the HFH. Obviously, if a domain has no hierarchical structure, then the HFH heuristic gives no meaningful contribution. The *hierarchy feature* of a flaw corresponds to the "independence" level of the timeline it belongs to. The idea is to exploit this hierarchy and select first flaws belonging to the most independent timeline. The underlying assumption is that the flaws influence is related to the corresponding timeline independence level. Consequently if "independent" flaws are solved first, i.e. flaws detected on one of the topmost timelines in the hierarchy, then we have a good probability to "automatically" solve or reduce the possible solutions of the "dependent" flaws. Flaw-level reasoning allows the solving process to integrate planning and scheduling steps. This means that at any point in the solving process the planner can make a planning choice by selecting a goal to solve, or a scheduling choice by selecting a scheduling threat to solve, and so on. Similarly to timelines we can give a "structure" to the solving process by assigning a priority to solving steps. Namely we define the *type feature* of flaws and select flaws according to their type. For example we can decide to solve first goals and then scheduling threats in order to force the solving process to take planning decisions before scheduling ones. In addition to the above features we can also specify the *degree feature* of a flaw which characterizes the criticality of a flaw. Similarly to the *fail first principle* in constraint satisfaction problems, the degree of a flaw is a measure of the number of solutions available to solve it. The lower is that degree, the higher is the criticality of the flaw, i.e. few options to solve the flaw. We use this feature to assign an higher priority to flaws with less possible solutions (i.e. more difficult to solve).

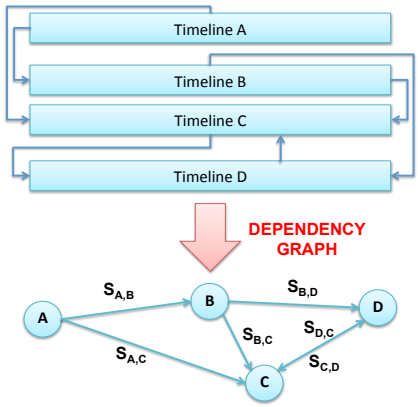

Fig. 1. A Dependency Graph example based on synchronizations.

The *Hierarchical Flaw-selection Heuristic* (HFH) we have defined, combines together alle the features described of the flaw in order to make the "best" choice for selecting the next flaw to solve during the solving process Given a set of flaws

Φ detected on a current plan π, HFH selects the best flaw to solve by applying a *pipeline* of filters that evaluate flaws according by considering the above features as follows:

$$\Phi^0(\pi) \xrightarrow{f_h} \Phi^1(\pi) \xrightarrow{f_t} \Phi^2(\pi) \xrightarrow{f_d} \Phi^3(\pi) \rightarrow \phi^* \in \Phi^3(\pi)$$

Every filter of the pipeline (f_h, f_t and f_d) filters plan flaws by taking into one of flaw feature described. Thus, the initial set of plan flaws $\Phi^0(\pi)$ is filtered by applying the filter f_h which returns the subset of flaws $\Phi^1(\pi) \subseteq \Phi^0(\pi)$ belonging to the most independent timelines. If no hierarchical structure can be found among domain timelines, then the filter f_h returns the initial set of flaws $\Phi^1(\pi) = \Phi^0(\pi)$, i.e. flaws are equivalent w.r.t. hierarchy feature. The filter f_t filters the set of flaws $\Phi^1(\pi)$ by taking into account the type feature of the flaws, e.g. f_t returns the subset of flaws containing only goals $\Phi^2(\pi) \subseteq \Phi^1(\pi)$. Finally the filter f_d filters the set of flaws $\Phi^2(\pi)$ by taking into account the degree feature of flaws and returns the final set $\Phi^3(\pi) \subseteq \Phi^2(\pi)$. The final set of flaws $\Phi^3(\pi) \subseteq \Phi^0(\pi)$ which results from the application of the pipeline, represents equivalent choices w.r.t. the heuristic point of view. Therefore HFH chooses the next flaw to solve ϕ^* by randomly selecting a flaw from the final set $\Phi^3(\pi)$the next flaw to solve is randomly selected from the final set $\phi^* \in \Phi^3(\pi)$.

4 Applying *Flexible Timeline-based Planning* to a Manufacturing Case Study

As a running example, let us consider a pilot plant from the on-going research project *Generic Evolutionary Control Knowledge-based mOdule* (GECKO): a manufacturing system for Printed Circuit Boards (PCB) recycling [17]. The objective of the system is to analyze defective PCBs, automatically diagnose their faults and, depending on the gravity of the malfunctions, attempt an automatic repair of the PCBs or send them directly to shredding. The pilot plant contains 6 working machines that are connected by means of a Reconfigurable Transportation System (RTS), composed of mechatronic components, i.e., transportation modules. Figure 2(a) provides a picture of a transportation module. Each module combines three transportation units. The units may be unidirectional and bidirectional units; specifically the bidirectional units enable the lateral movement (i.e., cross-transfers) between two transportation modules. Thus, each transportation module can support two main (straight) transfer services and one-to-many cross-transfer services. Figure 2(b) depicts two possible configurations.

Configuration 1 supports the forward (F) and backward (B) transfer capabilities as well as the left (LC1) and right (RC1) cross transfer capabilities. Configuration 2 extends Configuration 1 by integrating a further bidirectional transportation unit with cross transfer capabilities LC2 and RC2. The maximum number of bidirectional units within a module is limited just by its straight length (three, in this particular case). The transportation modules can be connected back to back to form a set of different conveyor layouts. The manufacturing

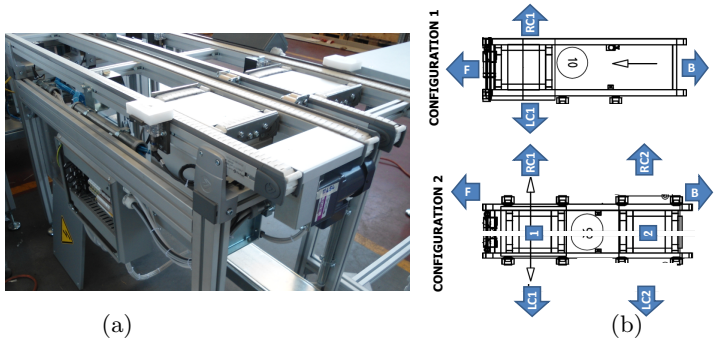

Fig. 2. (a) A transportation module; (b) Their transfer services.

process requires PCBs to be loaded on a fixturing system (pallet) in order to be transported and processed by the machines. The transportation system is to move one or more pallets and each pallet can be either empty or loaded with a PCB to be processed. Transportation modules control systems have to cooperate in order to define the paths the pallets have to follow to reach their destinations.

The description of the distributed architecture and some experimental results regarding the feasibility of the distributed approach w.r.t. the part routing problem can be found in [18,19]. *Transportation Module*s (TMs) rely on P&S technology to synthesize activities for supporting the work flow within the shop floor. Each TM agent is endowed with a Timeline-based planner (build on top of EPSL framework) to build plans for the transportation task.

4.1 The Gecko Timeline-based model

Figure 3 shows the timeline-based model of a generic *transportation module* (TM) of the GECKO case study. The timeline-based model has been defined by applying the modeling approach described in 2.1. Namely, a functional state variable *Channel* represents the high level transporting tasks of a TM. Each value of the Channel state variable models a particular transportation task indicating the corresponding input and output port of the module. For instance, *Channel_F_B* models the task of transporting a pallet from *port-F* to *port-B* w.r.t. Figure 2(b).

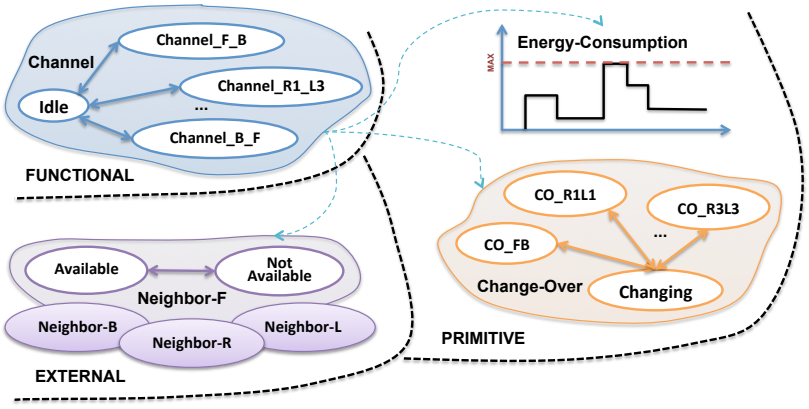

Fig. 3. Timeline-based model for a full instantiated TM

The *Change-Over* component is a primitive state variable which models the set of internal configurations the transportation module can assume for exchanging pallets with other modules. Namely configurations identify the internal paths a pallet can follow to traverse the module. For instance, *CO_F_B* in Figure 3 represents the configuration needed for transporting a pallet from port F to port B. The *Energy_Consumption* element in Figure 3 is a renewable resource component which models the energy consumption profile of a TM of the plant. A system requirement entails that the instant energy consumption of TMs cannot exceed a predefined limit for the physical device. This requirement is modeled by means of synchronization rules between the functional state variable and the renewable resource. These rules specify the energy consumption estimate for each functional activity of the module. For example, the maximum instant energy consumption allowed is 10 units, that *Channel_F_B* activity consumes 9 units of energy during its execution, *Channel_L1_R1*, *Channel_L2_R2*, *Channel_L3_R3* consume 3 units of energy during their execution and so on. Therefore the planner must schedule functional activities satisfying the energy consumption constraint.

A set of external state variables complete the domain by modeling possible states of TM's neighbor modules. Neighbors are modeled by means of external state variables because they are not under the control of the module. Namely, the TM cannot decide the state of its neighbors. However it is important to monitor their status because a TM must cooperate with them in order to successfully carry out its tasks. For instance the TM must cooperate with *Neighbor_F* and *Neighbor_B* to successfully perform a *Channel_F_B* task. Therefore *Neighbor_F* and *Neighbor_B* must be *Available* during task "execution".

Finally a set of synchronization rules specify how a TM implements its channel tasks (see the dotted arrows in Figure 3). According to the modeling approach described in 2.1, the synchronization specification follows a hierarchical decomposition. In this way they allow to specify a set of operative constraints (e.g. temporal constraints) describing the sequence of internal configurations and "external" conditions needed to safely perform channel tasks. For instance, a synchronization rule for the *Channel_F_B* task requires that the module must be set in configuration *CO_F_B* and that neighbor F and neighbor B must be *Available* during the "execution" of the task.

4.2 Experimental Evaluation

We have defined a set of timeline-based planning domain variants for the Gecko case study in order to evaluate Epsl solving capabilities. In particular we have modelled a generic TM in the plant considering several configurations in order to define scenarios of growing complexity. We have assumed that module's neighbor agents are always available and vary the number of cross transfers composing the module: (i) *simple* is the configuration with no cross transfers; (ii) *single* is the configuration with only one cross transfer; (iii) *double* is the configuration with two cross transfer; (iv) *full* is the configuration with three cross transfers (the maximum allowed w.r.t. the plant in our case study 4). The higher is the number of available cross transfers, the higher the number of elements and

constraints the planner has to deal with at solving time. Moreover, we defined several EPSL-based planners by varying the *heuristic function* applied during the solving process: (i) The *HFH* planner uses HFH; (ii) The *TFS* planner uses a heuristic based only on the $\mathcal{H}_{ft}$ feature; (iii) The *TLFS* planner uses a heuristic based only on the $\mathcal{H}_{fh}$ feature; (iv) The *DFS* planner uses a heuristic based only on the $\mathcal{H}_{fd}$ feature; (v) The *RFS* planner uses no heuristics at all, i.e. it makes a random selection of the *best* flaw to solve.

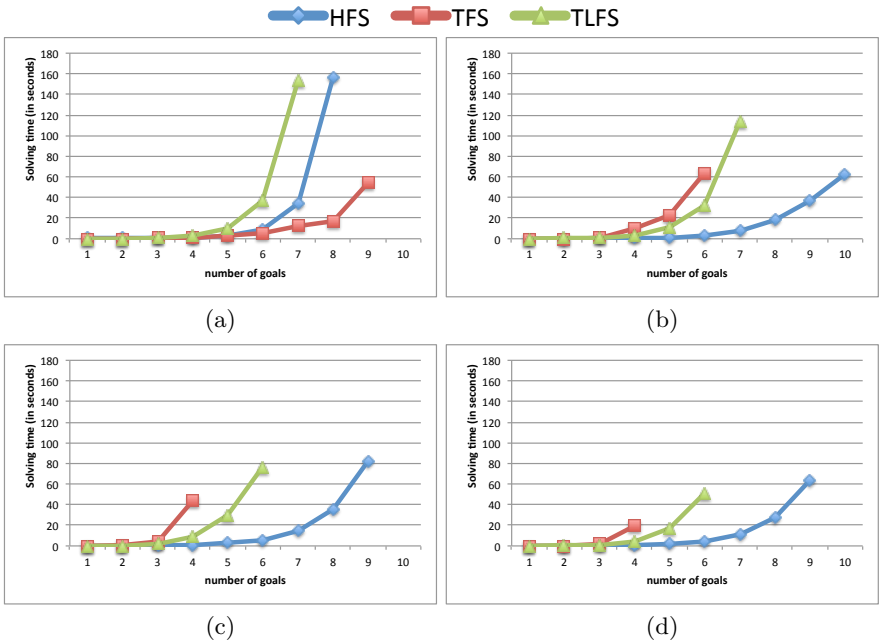

Fig. 4. Epsl-based planners performances on: (a) *simple* configuration; (b) *single* configuration; (c) *double* configuration; (d) *full* configuration

The charts in Figure 4 show the solving time trends of the EPSL-based planners (within a timeout of 180 seconds) w.r.t. the growing dimension of the planning problem (i.e. a growing number of goals) and the growing complexity of the module to control (i.e. the number of available cross transfers). The results show that the *HFH* planner is dominating other planners on the considered planning domains. Comparing the *TFS* planner (the original EPSL planner setting we used before the introduction of the timeline hierarchy) with the *HFS* planner, the deployment of HFH entails a general improvement of performance in terms of both solving times and scalability of the solving capacity of EPSL framework. The results concerning the *DFS* planner and the *RFS* one are not plotted in Figure 4 because these planners could not solve but the simpler cases.

It is worth pointing out that the *TFS* planner outperforms the *HFH* planner only in the *simple* domain, see Figure 4(a). This seems a consequence of the fact that the *TFS* planner maintains a "global" vision of the activities to

be performed during the solving process thus reasoning about the overall plan and taking into account all the timelines together. In the *simple* domain, the TM can perform channel tasks only towards two directions (*Channel_F_B* and *Channel_B_F*). Therefore, when building the plan, the planner can more suitably organize the tasks in order to reduce the number of reconfigurations of the module (i.e. change overs). Namely, that planner is able to "group" channel tasks requiring the same configuration of the module (e.g. scheduling all the *Channel_F_B* tasks before *Channel_B_F* tasks). This allows the planner to reduce the number of plan decisions simplifying its construction in the *simple* domain.

Conversely, the *HFS* planner maintains a "local" vision related to the single timelines of the domain because it builds one timeline at a time. Therefore, when that planner must manage the needed reconfiguration of the module (i.e. flaws on *Change Over* timeline), its choices are partially constrained by the channel timeline which has been built before the *Change Over* timeline following the timeline hierarchy. As a consequence, the planner is not able to organize tasks as in the case of *TFS* but it has to manage a larger number of tokens on the timeline at this point. However the *HFS* planner scales better than the *TFS* with the growing complexity of the domain as Figure 4 shows.

5 Conclusions and Future works

In this paper we have presented some recent advancements in the development of the EPSL framework. In particular, EPSL has been extended in two different aspects by introducing: (i) The capability of modeling and reasoning about renewable resources; (ii) The HFH heuristic to support the solving process. Finally, some experimental results have been reported in order to show the feasibility of EPSL-based planners when deployed to a real world domain.

The comparison of EPSL with different timeline-based P&S systems (e.g. EUROPA), the possibility to further extend the framework by integrating different type of resources (i.e., consumable resources) and the definition of new heuristics or improving HFH constitute future works in our research agenda. Moreover, one additional long-term research goal is to identify a set of quality metrics to characterize flexible timeline-based plans and, thus, exploit such metrics in order to generate better plans.

Acknowledgments. Andrea Orlandini is partially supported by the Italian Ministry for University and Research (MIUR) and CNR under the GECKO Project (Progetto Bandiera "La Fabbrica del Futuro").

References

1. Muscettola, N.: HSTS: Integrating Planning and Scheduling. In Zweben, M. and Fox, M.S., ed.: Intelligent Scheduling. Morgan Kauffmann (1994)
2. Jonsson, A., Morris, P., Muscettola, N., Rajan, K., Smith, B.: Planning in Interplanetary Space: Theory and Practice. In: AIPS-00. Proceedings of the Fifth Int. Conf. on AI Planning and Scheduling. (2000) 177–186

3. Cesta, A., Cortellessa, G., Denis, M., Donati, A., Fratini, S., Oddi, A., Policella, N., Rabenau, E., Schulster, J.: MEXAR2: AI Solves Mission Planner Problems. IEEE Intelligent Systems **22**(4) (2007) 12–19
4. Barreiro, J., Boyce, M., Do, M., Frank, J., Iatauro, M., Kichkaylo, T., Morris, P., Ong, J., Remolina, E., Smith, T., Smith, D.: EUROPA: A Platform for AI Planning, Scheduling, Constraint Programming, and Optimization. In: 4th Int. Competition on Knowledge Engineering for P&S (ICKEPS). (2012)
5. Ghallab, M., Laruelle, H.: Representation and control in IxTeT, a temporal planner. In: Proc. of the International Conference on AI Planning Systems (AIPS). (1994) 61–67
6. Cesta, A., Fratini, S.: The Timeline Representation Framework as a Planning and Scheduling Software Development Environment. In: Proc. of the 27^{th} Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG-08). (2008)
7. Cesta, A., Orlandini, A., Umbrico, A.: Toward a general purpose software environment for timeline-based planning. In: 20th RCRA International Workshop on Experimental Evaluation of Algorithms for solving problems with combinatorial explosion. (2013)
8. Cialdea Mayer, M., Orlandini, A., Umbrico, A.: A formal account of planning with flexible timelines. In: The 21st International Symposium on Temporal Representation and Reasoning (TIME), IEEE (2014) 37–46
9. Cimatti, A., Micheli, A., Roveri, M.: Timelines with temporal uncertainty. In: Proc. of the 27th AAAI Conference on Artificial Intelligence, AAAI Press (2013)
10. Cialdea Mayer, M., Orlandini, A.: An executable semantics of flexible plans in terms of timed game automata. In: The 22st International Symposium on Temporal Representation and Reasoning (TIME), IEEE (2015) To appear.
11. Orlandini, A., Suriano, M., Cesta, A., Finzi, A.: Controller synthesis for safety critical planning. In: 25th International Conference on Tools with Artificial Intelligence (ICTAI), IEEE (2013) 306–313
12. Py, F., Rajan, K., McGann, C.: A Systematic Agent Framework for Situated Autonomous Systems. In: Proc. of the 9^{th} Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS-10). (2010)
13. Bernardini, S.: Constraint-based Temporal Planning: Issues in Domain Modelling and Search Control. PhD thesis, Università degli Studi di Trento (2008)
14. Fratini, S., Pecora, F., Cesta, A.: Unifying Planning and Scheduling as Timelines in a Component-Based Perspective. Archives of Control Sciences **18**(2) (2008)
15. Borgo, S., Cesta, A., Orlandini, A., Umbrico, A.: An ontology-based domain representation for plan-based controllers in a reconfigurable manufacturing system. In: The 28th International FLAIRS Conference, AAAI (2015)
16. Cesta, A., Oddi, A., Smith, S.F.: A Constraint-based method for Project Scheduling with Time Windows. Journal of Heuristics **8**(1) (2002) 109–136
17. Borgo, S., Cesta, A., Orlandini, A., Rasconi, R., Suriano, M., Umbrico, A.: Towards a cooperative -based control architecture for a reconfigurable manufacturing plant. In: 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2014), IEEE (2014)
18. Carpanzano, E., Cesta, A., Orlandini, A., Rasconi, R., Valente, A.: Intelligent dynamic part routing policies in plug&produce reconfigurable transportation systems. CIRP Annals - Manufacturing Technology **63**(1) (2014) 425 – 428
19. Carpanzano, E., Cesta, A., Orlandini, A., Rasconi, R., Suriano, M., Umbrico, A., Valente, A.: Design and implementation of a distributed part-routing algorithm for reconfigurable transportation systems. International Journal of Computer Integrated Manufacturing. To appear. (2015)