Temporal Reasoning for a Collaborative Planning Agent in a Dynamic Environment*

Meirav Hadad
Department of Mathematics and Computer Science
Bar-Ilan University
Ramat-Gan 52900, Israel

Sarit Kraus
Department of Mathematics and Computer Science
Bar-Ilan University
Ramat-Gan 52900, Israel

Institute for Advanced Computer Studies, University of Maryland
College Park, MD 20742
ph.: 972-3-5318863

Yakov Gal
Division of Engineering and Applied Sciences
Harvard University
Cambridge, MA 02138 USA

Raz Lin
Department of Mathematics and Computer Science
Bar-Ilan University
Ramat-Gan 52900, Israel

March 18, 2002

Abstract

We present a temporal reasoning mechanism for an individual agent situated in a dynamic environment such as the web and collaborating with other agents while interleaving planning and acting. Building a collaborative agent that can flexibly achieve its goals in changing environments requires a blending of real-time computing and AI technologies. Therefore, our mechanism consists of an Artificial Intelligence (AI) planning

*This material is based upon work supported in part by the NSF, under Grant No. IIS-9907482.

subsystem and a Real-Time (RT) scheduling subsystem. The AI planning subsystem is based on a model for collaborative planning. The AI planning subsystem generates a partial order plan dynamically. During the planning it sends the RT-Scheduling subsystem basic actions and time constraints. The RT scheduling subsystem receives the dynamic basic actions set with associated temporal constraints and inserts these actions into the agent's schedule of activities in such a way that the resulting schedule is feasible and satisfies the temporal constraints. Our mechanism allows the agent to construct its individual schedule independently. The mechanism handles various types of temporal constraints arising from individual activities and its collaborators. In contrast to other works on scheduling in planning systems which are either not appropriate for uncertain and dynamic environments or cannot be expanded for use in multi-agent systems, our mechanism enables the individual agent to determine the time of its activities in uncertain situations and to easily integrate its activities with the activities of other agents. We have proved that under certain conditions temporal reasoning mechanism of the AI planning subsystem is sound and complete. We show the results of several experiments on the system. The results demonstrate that interleave planning and acting in our environment is crucial.

1 Introduction

Cooperative intelligent agents acting in uncertain dynamic environments should be able to schedule their activities under various temporal constraints. Temporal constraints may arise when an agent plans its own activities, or when an agent coordinates its activities with other collaborating agents.

Our work is based on the SharedPlan model of collaboration [23] that supports the design and construction of collaborative systems. It includes planning processes that are responsible for completing partial plans, for identifying recipes, for reconciling intentions, and for group decision making. Determining the execution times of the single-agent and multi-agent actions in SharedPlans is difficult because actions of different agents must be coordinated, plans are often partial, knowledge of other agents' activities and of the environment is often partial, and temporal and resource constraints must be accommodated.

In this paper, we present a mechanism that enables cooperative agents to interleave planning for a complex activity with the execution of the constituents of that activity. Each agent reasons individually while interacting with other agents. Each agent dynamically determines the durations and time windows for all the actions it has to perform in such a way that all of the appropriate temporal constraints of the joint activity will be satisfy. The execution times and the durations of the constituent activities need not be known in advance. That is, an agent may change its timetable easily if it identifies new constraints arising from changes in the environment or communication with other agents. Furthermore, if the agent determines that the course of action it has adopted is unsuccessful, then it can easily revise its timetable of future actions. The mechanism in this paper focuses on temporal scheduling. Thus, to simplify, the planner does not take into consideration preconditions and effects.

Our mechanism's ability to schedule an agent's actions under uncertainty contrasts with other planners, who rely on perfect domain knowledge throughout plan development and execution [34, 16, 11, 4]. Others, who can plan under uncertainty, are complex and cannot

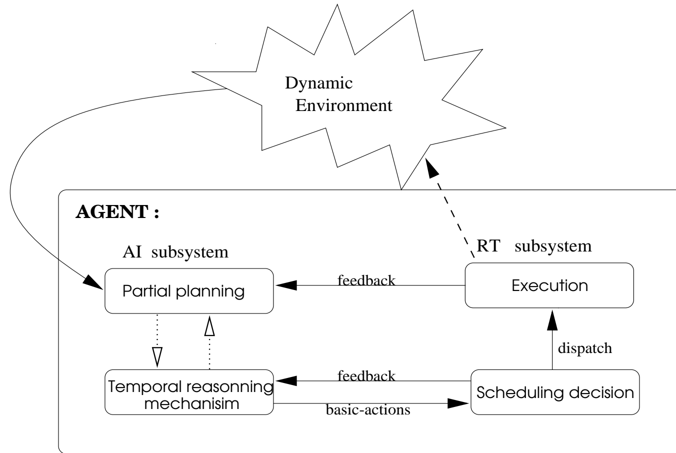

Figure 1: The structure of individual agent in the system.

be easily extended for use in cooperative multi-agent environments [70, 19]. Our mechanism is simple and appropriate for uncertain, dynamic, multi-agent environments, and enables agents to reason about their timetables during their planning process and thus to interleave planning and acting.

Though our mechanism is appropriate for collaborative multi-agent environments, it enables each agent to determine its own timetable independently. Thus, unlike other collaborative multi-agent systems, which either suggest broadcasting messages among team members to maintain the full synchronization [59, 65] or suggest that a team leader be responsible for determining the timing of the individual actions [27], our mechanism does not restrict the activity of the individual agents.

As will be described in section 2.1, building a collaborative agent that can flexibly achieve its goals in changing environments requires a blend of real-time computing and AI technologies [44]. Thus, our system consists of an AI planning subsystem and a Real-Time (RT) scheduling subsystem. Figure 1 illustrates the structure of an individual agent in our system. Given an agent's individual and shared goals, the AI planning subsystem plans the agent's activities. It computes incremently a set of basic actions[1] and temporal constraints on them. Before sending the basic actions to the RT scheduling subsystem, the AI planning subsystem determines consistency of the various types of the temporal constraints. The RT scheduling subsystem determines the exact time in which the basic actions will be performed in such a way that the constraints are satisfied.

In the next section we survey the related research fields on temporal reasoning and scheduling, and we compare these fields with our work. The SharedPlan model that is the basis of the AI planning subsystem is briefly described in section 3. The temporal reasoning algorithm of the AI planning subsystem for the individual case is presented in

[1]We define a basic action as an action which must fulfill three conditions: (a) it does not involve more than one agent; (b) it must be performed in one sequence without preemption; and (c) there is only one way to perform the action.

section 4. In section 5 we display the theorem of the correctness of this reasoning algorithm and discuss its complexity. Then, in section 6, we present a heuristic algorithm for scheduling by the RT scheduling subsystem. This heuristic tries to find a feasible schedule in which all tasks meet their appropriate deadlines. In section 7 we present the results of several experiments performed on our system. We conducted these experiments in order to evaluate the performance of the system and to study the influence of several parameters on the system's performance. Finally, in section 8, we show how to expand the individual case into the multi-agent case.

2 Background

2.1 AI vs RT Systems

Much traditional and current AI research revolves around building powerful search-based planning mechanisms that can find useful plans of action in complex domains that include goal interactions, uncertainty and temporal information [65, 23, 36, 30, 43]. Unfortunately, the traditional AI systems have been developed without much attention to critical temporal limitations that motivate RT systems research. However, as these AI systems move to real-world applications, they also become subject to the temporal constraints of the environments in which they operate. Thus, the needs of real applications are generating the integration of the RT and AI system design technologies. This section discusses the issues arising from attempting to combine RT methods and AI methods.

A real-time AI problem solver must operate under certain temporal constraints imposed by the environment. The control system of a real-time AI problem solver must perform its search process in such a way that the temporal constraints of the problem are satisfied. Real-time AI problem solvers differ from conventional RT systems. For example, in conventional real-time domains, the system must know all the tasks that need to be executed, as well as their worst-case resource requirements before they can be completed. In addition, the system must be able to finish building the schedule of the tasks before any of the tasks begin to be executed. Real-time AI problem solvers aim at dealing with other constraints of the environment, such as uncertainty and incomplete knowledge about the environment, dynamics in the world, bounded validity time of information, and other resource constraints [25].

Musliner et al. [44] compare the features of RT and AI systems and present a table which summarizes these features (see columns 2 and 3 in Table 1). As we can see in this table, their comparison reveals a conflict between the constraints involved in making RT guarantees and the characteristics of traditional AI methods.

The characteristics of RT systems (e.g., [32, 71, 9, 56, 62, 72, 39]) are summarized in the second column of Table 1: an RT system assumes an environment that may be dynamic (in the sense that the tasks required may vary at run time), but at least has knowledge of the worst-case task requirements (e.g., the deadline, release time, and duration of the task). Most RT systems run numeric control algorithms with well-understood resource requirements and performance. Using these worst-case measures, it is possible to build task schedules that allocate a system's limited execution resources and provide guaranteed response times. Thus,

	Real-Time system	**Traditional AI planner**	**Our system**
Environment	dynamic, known worst-case	static, closed-world, predictable	dynamic, uncertain, incomplete information
Tasks	classical control, numeric algorithms	search, look ahead planning	partial planning
Resources	limited	assumed sufficient	limited
Response time	guaranteed	high-variance or unbounded	high-variance planning, guaranteed

Table 1: Comparing features of AI and RT systems.

RT research has focused on developing methods that guarantee that a particular set of tasks can be executed under domain's temporal constraints [63, 40, 6].

The third column of Table 1 outlines characteristics of traditional AI planning systems and reveals a sharp contrast with RT systems. Most AI systems (e.g.,[26, 69]) are based on the "closed world" assumption: the AI-controlled agent is the only source of change in the world. Within this environment, the AI system's task is to plan some future course of action using projection (look ahead) and search. Most planners assume that the agent executing the plan will have essentially unlimited sensing and processing resources [50, 54, 55, 7]. As a result, the time needed to find a plan in the worst case scenario may be several orders of magnitude longer than the average case. This means that allocating resources to guarantee the worst-case response time will be very costly and will lead to very low utilization of a system's resources. Furthermore, AI systems with powerful knowledge representation or learning abilities ([17, 64]) may have unbound worst-case response times. In these cases, it is impossible to allocate sufficient resources in advance, and thus RT guarantees are unfeasible.

The fourth column in this table, which combines characteristics from both domains, describes the characteristics of our system. In our system, which is based on the SharedPlan model, the agent has a description of its goals, its environment, and its possible actions. The system uses partial planning methods to choose the correct action for any particular situation within the unbound world model. These agents, which act in dynamic environments, are uncertain about their own actions and have incomplete information about other agents and the environment. That is, the worst case requirements of their tasks may be unknown. Furthermore, agents which act in a cooperative environment may have limited resources. In addition, the agents must plan their activities under constraints and provide a guaranteed response time.

To meet these challenges, we combine AI and RT techniques into a single system. Musliner et al. [46] describe three fundamental approaches for integrating AI and RT systems. The first approach forces AI computation to meet deadlines identical to other real-time tasks. The goal, then, is to be "intelligent in real time." For example, Hamidzadeh and Shekhar [25] use this approach in their DYNORAII real-time planning algorithm. As we will describe in section 6, the scheduling problem that our RT scheduling subsystem faces is NP-complete. Thus, in some cases, the RT scheduling subsystem finds a feasible schedule

by using the simulated annealing algorithm [55], which is a well known search method from the AI field.

The second approach essentially assumes that the overall system employs typical AI search-based deliberation techniques, but under certain circumstances these techniques will be short-circuited in favor of real-time reflexive action. This type of system is suitable for domains where deliberative action is the norm and mission-critical, real-time reactions are rare. The Soar system [33] is one example of these types of systems.

The third approach, which has motivated the design of our system, tries to retain each system's strength by allowing separate RT and AI subsystems to cooperate in achieving overall desirable behaviors. The subsystems must be isolated so that the AI mechanisms do not interfere with the guaranteed operations of the RT subsystem, but the subsystems must also communicate and judiciously influence each other. Thus, cooperative systems can be seen as being "intelligent about real time," rather than "intelligent in real time." Examples of such architectures include Arkin's Autonomous Robot Architecture (AuRA) [3], Simmons' Task Control Architecture (TCA) [57], Miller and Gat's three-layer ATLANTIS system [41], and the CIRCA system [45].

However, in all of these systems, the RT subsystem is a reactive system that is embedded in the dynamic environment and responds by taking actions that affect that environment without using any planning on reasoning methods. However, when it facing a problem (for example, a mobile robot facing an obstacle), the job of the AI subsystem is to resolve the problem by building a plan and sending the constituent primitive actions to the RT subsystem. That is, the AI subsystem in these works does not include any ability to reason about the temporal constraints of the high-level actions and the primitive actions in the problem. As a result, although these systems work in uncertain and dynamic environments, the AI subsystem works as a traditional AI problem solver which is appropriate only for "closed world" environments. In our work, the AI-P subsystem is embedded in the dynamic environment and interacts with it. Our AI-P subsystem enables the agent to commit to goals as well as to actions that will enable it to achieve those goals under several kinds of temporal constraints. It then builds a partial plan to achieve those goals. Each primitive action in the plan is sent to the RT scheduling subsystem for execution under appropriate temporal constrains.

An additional example of a system that belongs to the third approach is the RealPlan system [60]. This system consists of a planner and a scheduler. The planner uses traditional AI methods for reasoning about the way to allocate resources. After the planning is complete, the scheduler decides which resources to actually allocate based on resource allocation policies proposed by the planner. The resource allocation polices proposed by the planner are given in terms of constraints on values of scheduling variables. The RealPlan system does not produce an exact timetable, but, rather, merely determines the order of the execution of the planned actions in such a way that 2 actions will not use the same resource during the same interval. Thus, they do not consider critical temporal limitations as we do.

2.2 Temporal Reasoning and Scheduling in AI Systems

As mentioned above, contrary to our dynamic AI environment, the RT subsystem assumes that the worst-case task requirements are known. Namely, the task data in an RT system consists of its performance requirements, including specific deadlines, precedence constraints, and duration time. However, in our uncertain dynamic environment, these performance requirements are typically unknown. Thus, we suggest that the AI planning subsystem must be able to reason about incomplete knowledge. That is, the AI planning subsystem must be able to reason about the temporal requirements of basic actions. When the AI planning subsystem sends a basic action to the RT scheduling subsystem, it provides temporal requirements as well.

Representing and reasoning about incomplete and indefinite qualitative temporal information is an essential part of many AI systems for individual agents. Several formalisms for expressing and reasoning about temporal knowledge have been proposed, most notably, Allen's interval algebra [2], Vilain and Kautz's point algebra [68], and Dean and McDermott's time map [13]. Each of these representation schemes is supported by a specialized constraint-directed reasoning algorithm. At the same time, extensive research has been carried out on problems involving general constraints as in [42]. Some of these have been extended to problems involving temporal constraints [14, 5]. Since all of these algorithms require their input to include all the constraints on the events, and since these constraints cannot be changed during the run of the algorithms, these works can be applied only in "static" environments in which the occurrence times of events and their durations are known beforehand.

Recently, several works have developed techniques for "real world" environments, taking into account changes in the environment while executing a plan. Although they suggest an intelligent control system that can dynamically plan its own behavior, they do not take temporal constraints into consideration. Examples of such works include M-SHOP [49] which is focused on domain-independent planning formalization and planning algorithms; the Zeno system [29], which suggests a method for building a decision-making mechanism for a planner in an uncertain environment; and the SGP contingent planning algorithm [12], which handles planning problems with uncertainty in initial conditions and with actions that combine causal and sensory effects. It also includes the planning model of the constraint-based EXCALIBURE planning system [47, 48] and so on. The main goal of this paper is to handle temporal constraints of the activity.

Other recent planners such as O-plan [11], ZENO [51], ParcPlan [16, 34] and Cypress [70] are able to handle temporal constraints. However, they do not produce an exact timetable, but rather only determine the order of the execution of the planned actions. Thus, most of them (e.g., [11, 51, 16, 34]) do not interleave planning and execution. Therefore they cannot, for example, backtrack in case of failure during execution. Even though Cypress [70] is built from a planning subsystem (i.e., SIPE-2) and an execution subsystem (i.e., PRS-CL) and therefore interleaves planning and execution, its execution subsystem is unable to handle temporal constraints. As a result, this system cannot perform planning and scheduling, as does our system. In addition, since these works do not determine explicit times for the planned actions, using such systems in a cooperative multi-agent environment is problematic.

Vidal and Ghallab [67] extend classical temporal constraint networks to handle all the

types of temporal constraints presented by Allen [1] in uncertain environments. While they handle a wider range of constraints than we do, they do not study how their mechanism can be used by a planner.

In other works that combine planning and scheduling methods [8, 66, inter alia], the planner builds a complete plan of the actions that it intends to perform before it begins executing any of them. Also, in most cases, both the duration and the time window of each action that the planner needs to schedule are known in advance. These restrictions and requirements are not needed in our mechanism's applications.

In our work, we use networks of binary constraints [42], which are special cases of a general class of problems known as *constraint satisfaction problems*. Our development extends the previous works on temporal reasoning, as in [14]. However, in contrast to these previous works, our algorithm is also appropriated for distributed, dynamic, and uncertain environments. The AI planning subsystem in our work consists of a component for determining consistency of temporal data and procedures for discovering or inferring new facts about this data. Our planning system is based on the SharedPlan model for collaborative agents. Using this model enables us to build reasoning mechanisms for collaborative multi-agent environments. The building of the temporal reasoning component is difficult because the SharedPlan model deals with agents that may have only partial knowledge of the way in which to perform an action.

2.3 Scheduling in RT systems

As we mentioned in section 2.1, several works [44, inter alia] propose a blending of real-time computing and AI technologies in which a RT system is responsible for the execution of the actions generated by a planner. However, the RT systems in these works are based on existing real-time developments. As a result, they do not pay attention to the unique properties of the tasks which are generated by the agent and are associated with uncertainty. In the literature on real-time, computing a schedule at run time is often called "dynamic scheduling," in contrast to pre-run or "static scheduling," which assumes complete knowledge of all tasks before run time. Because our system must meet hard timing and precedence constraints, the predictability of the system becomes an important concern.

Most existing RT scheduling systems rely on pre-run static scheduling to achieve this, because computing the schedule at run time cannot, in general, guarantee that a feasible schedule is found [73]. For example, Xu and Parnas [72] present a static scheduling algorithm that finds an optimal schedule on a single processor for a given set of processes with arbitrary release times, deadlines, precedence and exclusion relations, through a branch and bound algorithm. The computation time to produce the schedule they present grows exponentially as the problem size increases. Most research dealing with dynamic scheduling of tasks has focused on preemptive scheduling or periodic task sets ([58] inter alia), or distributed systems [53, inter alia], both of which do not match the requirements of task sets generated by our cooperative intelligent agents, which are non-preemptive and a-periodic. Other works [21] do not consider precedence relations between the tasks, which is considered to be a very important requirement in the AI environment of a planner, in which several actions are preconditions of other actions. Feasible dynamic scheduling in such an environment has

been classified as an NP-Hard problem [18]. A practical scheduling algorithm must be based on heuristics that are practical and adaptive.

Stankovic and Ramamritham [61] have developed the Spring Algorithm, which schedules non preemptive tasks at run-time. The Spring system treats the scheduling problem as a search tree, and directs each scheduling move to a plausible path by choosing the task with the smallest possible value of a heuristic function. Whenever a task is added to the partial schedule and deemed infeasible, backtracking must be performed in order to find a different task to append to the partial schedule which might lead to a feasible schedule. This algorithm leads to good results for domains such as operating systems, in which several processes compete for a set of resources. We tried this algorithm in our domain; however, in our domain this algorithm provided poor results.

As will be described in the following section, the tasks in the set which are sent to the RT scheduling subsystem are the constitutes of a complex action that was separated into its basic components. Thus, the constraints to which the high-level action was subjected will also influence the constraints of its constitutes. As a result, all of the basic actions which are constitutes of a specific complex action must be scheduled in the time interval of this complex action. Thus, it is probable that if we build an initial schedule by using the "Earliest Deadline First" (EDF) algorithm, we will attain an initial schedule which will be close to the desirable schedule. This factor led to the development of our own scheduling algorithm, which consists of two major parts. The first is a primary EDF scheduler. The second is a simulated annealing heuristic which which has been proved to be efficient for other scheduling problems [15, inter alia]. The algorithm is described in section 6.

3 The SharedPlan Model

When agents form teams, new problems emerge regarding the representation and execution of joint actions. A team must be aware of and concerned with the status of the group effort as a whole. To rectify this problem, it was proposed that agents have a well-grounded and explicit model of cooperative problem solving on which their behavior can be based. Several such models have been proposed [35, 30, 27], including the SharedPlan model [23], which is the basis of our work.

The SharedPlan formalization [23, 22] provides mental-state specifications of both *shared plans* and *individual plans*. SharedPlans are constructed by groups of collaborating agents and include subsidiary SharedPlans [36] formed by subgroups as well as subsidiary individual plans formed by individual participants in the group activity. The full group of agents must mutually believe that the subgroup or the agent of each subact has a plan for the subact. However, only the performing agent(s) itself needs to hold specific beliefs about the details of that plan.

Actions in the model are abstract complex entities that have been associated with various properties such as action type, agent, time of performance, and other objects involved in performing the action. Following Pollack [52], the model uses the terms "recipe" and "plan" to distinguish between knowing how to perform an action and having a plan to perform the action. When agents have a SharedPlan to carry out a group action, they have certain

individual and mutual beliefs about how the action and its constituent subactions are to be implemented.

The term *recipe* [52, 37] is used to refer to a specification of a set of actions, which is denoted by β_i $(1 \leq i \leq n)$, the performance of which under appropriate *recipe-constraints*, denoted by ρ_j $(1 \leq j \leq m)$, constitutes performance of α.[2] The meta-language symbol R_α is used in the model to denote a particular recipe for α.The subsidiary actions β_i in the recipe for action α, which are also referred to as a subact or subactions of α, may be either *basic actions* or complex actions.

Basic actions are executable at will if appropriate situational conditions hold. A complex action can be either a single-agent action or a multi-agent action. In order to perform some complex action, β_i, the agents have to identify a recipe R_{β_i} for it. There may be several recipes, R_{β_i}, for β_i. The recipe R_{β_i} might include constituent subactions δ_{iv}. The δ_{iv} may similarly be either basic or complex. Thus, the general situation considering the actions without the associated constraints ρ_j, is illustrated in Figure 2, in which the leaves of the tree are basic actions. We refer to this tree as "a complete recipe tree for α." The SharedPlan formalism uses **Select_Rec** and **Select_Rec_GR** to refer respectively to the planning actions that agents perform individually or collectively to identify ways to perform (domain) actions by extending the partial recipe R^p_α for α. This hierarchical task decomposition method of the partial order planning is known in the literature of AI planning systems as (HTN)-style [28].

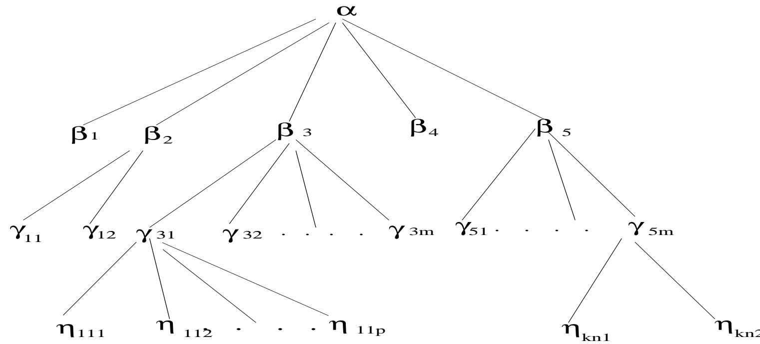

Figure 2: Recipe tree. The leaf nodes are basic actions.

Figure 3 presents an example of a possible recipe for some complex level action α. As shown in this figure, the recipe structure consists of subactions, temporal constraints, and may also include other entities. Each subaction may be either an individual action or a multi-agent action and is associated with temporal intervals; each interval represents the time period during which the corresponding subaction is performed. The recipe includes two types of temporal constraints, *precedence* constraints and *metric* constraints. The parameters of an action may be partially specified in a recipe and in a partial plan. For the agents to have achieved a complete plan, the values of the temporal parameters of the actions that

[2]The indices i and j are distinct; for simplicity of exposition, we omit the range specifications in the remainder of this document.

```
(make-recipe :action-type 'α
  :name                        'R_α^1
  :time-period                 'unknown
  :subactions                  (1) (β1 Agent1 T1 ···)
                               (2) (β2 Agent1 T2 ···)
                               (3) (β3 (Agent1 Agent2) T3 ···)
                               (4) (β4 Agent1 T4 ···)
                               (5) (β5 (Agent1 Agent2) T5 ···)
  :precedence-constraints      β1 before β2      β1 before β3
                               β2 before β4      β2 before β5
                               β3 before β5
  :metric-constraints       (finish-time T2− start-time T1 ≤ 20minutes),
                            (start-time T4− finish-time T2 ≤ 40minutes),
                            (finish-time T5− start-time T3 ≤ 60minutes),
                            (start-time T3 after 5:00)
                       ...
```

Figure 3: Recipe of the Complex Action α.

constitute their joint activity must be identified in such a way that all of the appropriate constraints are satisfied.

The problem with reasoning about the temporal parameters of the actions that the agent is committed to perform results from the dynamic decomposition of the actions in the SharedPlan model. When a high-level action is broken up into sequences of subactions and finally into basic actions, the time available to achieve the high-level action must also be split into intervals for each subaction. Doing this correctly would require the SharedPlan system to have a predictable mechanism of how long it takes to solve the subactions. Unfortunately, building such a predictable model is difficult because the SharedPlan model deals with agents that may only have partial knowledge on the way in which to perform an action. That is, as a result of the dynamic nature of plans, any of the components of its plan may be incomplete and the agent does not know the duration of the complex actions before it finishes constructing its plan. In this paper we describe the technique of temporal reasoning mechanisms which we propose to use in our system in order to build this type of predictable mechanism which may reason in a dynamic fashion.

In the following section we present a temporal reasoning algorithm that enables the identification of the values of the temporal parameters. First the mechanism for an individual agent is presented; then we describe how it can be expanded into a collaborative environment.

4 The Algorithm for Temporal Reasoning

To execute an action α, an agent has to execute all of the basic actions in a complete recipe tree for α under the appropriate constraints. The goal of the temporal reasoning algorithm of the AI planning subsystem is to develop a complete recipe tree and to find the temporal requirements associated with the basic actions. However, initially an agent may not be able to develop the entire recipe tree and to identify its constraints: it may only have partial knowledge about how to perform an action; it may have incomplete information about the

environment and other agents; and it may have to wait to receive temporal constraints of other agents. For example, the agents may only have a partial recipe for the action; or they may not yet have decided who will perform certain constituent subactions and therefore they may have no individual or collaborative plans for those acts; or, an agent may not have determined whether potential new intentions are compatible with its current commitments and if they can be adopted. As the agents reason individually, communicate with one another, and obtain information from the environment, portions of their plans become more complete. Furthermore, it may need to start executing some of the basic actions before it has been able to construct the entire tree. In addition, if an agent determines that the course of action it has adopted is not working or receives a new temporal constraint from another agent, then the recipe tree may revert to a more partial state. The AI planning subsystem enables the agent to construct the recipe tree incrementally and to backtrack when needed.

As soon as it is identified, each basic action β is sent to the RT scheduling subsystem[3] along with its temporal requirements $\langle D_\beta, d_\beta, r_\beta, p_\beta \rangle$, where D_β is the *Duration time*, i.e., the time necessary for the agent to execute the basic action β without interruption; d_β denotes the *deadline*, i.e., the time before the performance of the basic action should be completed; r_β refers to the *release time*, i.e., the time at which the basic action becomes ready for execution; p_β is the *predecessor actions*, i.e., the set $\{\beta_j | (1 \leq j \leq n)\}$ of basic actions whose execution must end before the beginning of the execution of β. Note that a basic action may be performed before the agent completes its plan to perform α.

Let $\mathcal{A} = \{\beta_i | (1 \leq i \leq m)\}$ be the set of all the basic actions which were sent dynamically to the RT scheduling subsystem as part of the performance of action α. It is important to note that the AI planning subsystem's algorithm does not check if there is a schedule for $\mathcal{A}$ that satisfies the temporal constraints and meets all the deadlines, since this problem is NP-complete [18]. However, the algorithm ensures that performing β_is under the associated constraints will constitute performing α without a conflict. Thus, a feasible schedule exists if actions can be performed in parallel. The task of finding a feasible schedule (if such a schedule exists) is left for the RT scheduling subsystem. If the RT scheduling subsystem fails to schedule certain basic actions, it informs the AI planning subsystem of its failure. However, because the AI planning subsystem ensures that the temporal requirements of all the basic actions do not conflict, the agent may ask another agent to execute the problematic basic action in parallel with other actions.

The temporal reasoning mechanism is based on previous work on the *temporal constraint satisfaction problem (TCSP)* [14, inter alia]. Formally, TCSP involves a set of variables, $X_1, \ldots, X_n$, having continuous domains[4]; each variable represents a time point. Each constraint is represented by a set of intervals: $\{I_1, \ldots, I_m\} = \{[a_1, b_1], \ldots, [a_m, b_m]\}$. A unary constraint, T_i, restricts the domain of variable X_i to the given set of intervals; namely, it represents the disjunction $(a_1 \leq X_i \leq b_1) \vee \cdots \vee (a_m \leq X_i \leq b_m)$. A binary constraint, T_{ij}, constrains the permissible values for the distance $X_j - X_i$; it represents the disjunction $(a_1 \leq X_j - X_i \leq b_1) \vee \cdots \vee (a_m \leq X_j - X_i \leq b_m)$.

A *network of binary constraints* (a binary TCSP) consists of a set of variables, $X_1, \ldots, X_n$, and a set of unary and binary constraints. Such a network can be represented by a *directed*

[3]Our RT scheduling subsystem can handle only basic actions.

[4]In our mechanism we do not force the assumption that the domain is continuous.

constraint graph, where nodes represent variables and an edge (i, j) indicates that a constraint T_{ij} is specified; it is labeled by the interval set. Each input constraint, T_{ij}, implies an equivalent constraint T_{ji}; however, only one of them will usually be shown in the constraint graph. A special time point, X_0, is introduced to represent the "beginning of the world." All times are relative to X_0. Thus we may treat each unary constraint T_i as a binary constraint T_{0i} (having the same interval representation). A tuple $X = (x_1, \ldots, x_n)$ is called a *solution* if the assignment $\{X_1 = x_1, \ldots, X_n = x_n\}$ satisfies all the constraints. The problem is consistent if at least one solution exists.

The general TCSP problem is intractable, but there is a simplified version, *simple temporal problem* (STP), in which each constraint consists of a single interval. This version can be solved by using the efficient techniques available for finding the shortest paths in a directed graph with weighed edges such as Floyd-Warshall's all-pairs-shortest-paths algorithm[5] [10]. In our work we also use a temporal constraints graph, which consists of the temporal constraints associated with action α (including precedence constraints and metric constraints). However, because of the uncertainty and dynamic environment of the agents, in contrast to previous works, the constraints graph which is built by our agents may include only partial knowledge; i.e., our algorithm enables the agents to build this graph incrementely and to backtrack when needed. In addition, the agents are able to determine the action for which the temporal parameters are known and to execute them.

4.1 Definitions and Notations for the Temporal Reasoning Mechanism

In order to present our technique for temporal reasoning, we will first define some basic concepts that will be used throughout our reasoning mechanism.

Definition 4.1 (Time interval of an action) *Let α be an action. We denote the time interval of α by $[s_\alpha, f_\alpha]$, where s_α is the time point at which the execution of action α starts and f_α is the time point at which the execution of action α ends.*

There are two types of temporal constraints in the system. The first type is *metric*, where we treat the duration of an event in a numeric fashion. The temporal range of the starting point and finishing point, the length of time between disjoint events, and so forth, are numbers which may satisfy specific inequalities or various measurable constraints. For example, "β_2 has to start at least 20 minutes after β_1 would terminate." The second type is *relative*, in which events are represented by abstract time points and time intervals; i.e., the temporal constraints make no mention of numbers, clock times, dates, duration, etc. Rather, only qualitative relations such as *before*, *after*, or *not after* are given between pairs of events; e.g., "the subaction β_1 has to occur before β_2." The techniques which are used to reason about these two types of constraints are different [20]. In the following we give more details about the usage of these types of constraints in our system and describe them formally.

As illustrated in figure 3, in certain cases the subactions, $\beta_1, \ldots, \beta_n$, of a recipe R_α have to satisfy certain precedence relations which are relative constraints. The precedence relationships between the subactions is specified in our system using events. An event represents

[5]Floyd-Warshall's algorithm efficiently finds the shortest paths between all pairs of vertices in a graph.

a point in time that signifies the completion of some actions or the beginning of new ones. The time points at which execution of action α starts and finishes are thus described by two events. In the system, events are represented by nodes, and the activity of action β_i is represented by a directed edge between the time points at which the execution of action β_i starts and finishes. A directed edge between the finishing time point of an action β_i and the starting time point of another action β_j, denotes that the execution of β_j cannot start until the execution of β_i has been completed. The label of the edge needs to be proportional to the duration either of the activity of some action β_i; or the delay between two different actions β_i and β_j, and is referred to as the *temporal distance* between them [38]. As we will see in the remainder of this section, this graphical representation will enable us to build the temporal network (TCSP) which is mentioned in the earlier section. We have termed this graphical form of the precedence relations the *precedence graph*. The formal definition of the precedence graph which we use in our SharedPlan system is presented in the following definition:

Definition 4.2 (Precedence graph of α, $Gr^p_{R_\alpha}$) *Let α be a complex action, and let R_α be a recipe which is selected for executing α. Let $\beta_1, \ldots, \beta_n$ be the subactions of the recipe R_α and $\theta^p_{R_\alpha} = \{(i,j) | \beta_i < \beta_j; i \neq j\}$ are the precedence constraints associated with R_α. The* precedence graph of α, *$Gr^p_{R_\alpha} = (V^p_{R_\alpha}, E^p_{R_\alpha})$ with reference to R_α and its precedence constraints $\theta^p_{R_\alpha}$ satisfies the following:*

1. *There is a set of vertices $V^p_{R_\alpha} = \{s_{\beta_1}, \ldots, s_{\beta_n}, f_{\beta_1}, \ldots, f_{\beta_n}\}$ where the vertices s_{β_i} and f_{β_i} represent the start time and the finish time points of β_i $1 \leq i \leq n$, respectively*[6].
2. *There is a set of edges $E^p_{R_\alpha} = \{(u_1, v_1), \ldots, (u_m, v_m)\}$, where each edge $(u_k, v_k) \in E^p_{R_\alpha}$ $1 \leq k \leq m$ is either:*
   (a) *the edge $(s_{\beta_i}, f_{\beta_i})$ that represents the precedence relations between the start time point s_{β_i} and the finish time point f_{β_i} of each subaction β_i. The edge is labeled by the time period for the execution of the subaction β_i; or*
   (b) *the edge $(f_{\beta_i}, s_{\beta_j})$ which represents the precedence relation $\beta_i < \beta_j \in \theta^p_{R_\alpha}$, which specifies that the execution of the subaction β_j starts after the execution of the subaction β_j ended. This edge is labeled by the delay period between β_i and β_j.*

All of the edges of the precedence graph are initially labeled by $[0, \infty]$.

An example of precedence graph is given in the following:

Example 1 :
Figure 4 illustrates a precedence graph $Gr^p_{R_\alpha}$=($V^p_{R_\alpha}$,$E^p_{R_\alpha}$). In this Figure the subactions of the selected recipe R_α are β_i $(1 \leq i \leq 5)$, and the precedence relations are $\theta^p_{R_\alpha} = \{\beta_1 < \beta_2, \beta_1 < \beta_3, \beta_2 < \beta_4, \beta_2 < \beta_5, \beta_3 < \beta_5\}$. These precedence relations are appropriated to the precedence constraints in the recipe of α, which is shown in figure 3.

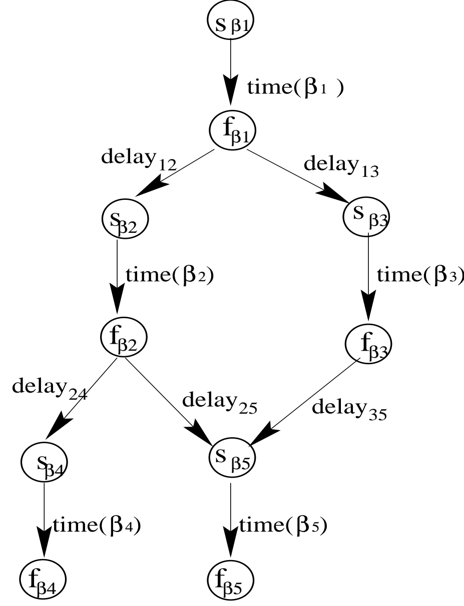

Figure 4: A precedence graph $Gr^p_{R_\alpha}$.

As depicted in Figure 4, the precedence relations do not induce a total order on all the subactions. For example, the agent can begin the execution of action β_3 before the completion of action β_2 and vice versa. In such cases these actions can be executed also in parallel: e.g., agent G_1 can begin to perform action β_2 and another agent, G_2, can begin to perform action β_3 in the same time interval.

Definition 4.3 (parallel actions) *Let $Gr^p_{R_\alpha}$=($V^p_{R_\alpha}$,$E^p_{R_\alpha}$) be the precedence graph of an action α. Let β_i and β_j, $i \neq j$, be two subactions in R_α. The subactions β_i and β_j are called* parallel actions *if $Gr^p_{R_\alpha}$ does not include a path either from s_{β_i} to s_{β_j} or from s_{β_j} to s_{β_i}, i.e. there is no precedence constraints between β_i and β_j.*

In Figure 4, actions β_3 and β_2 are parallel actions, as are actions β_4 and β_5.

Definition 4.4 (beginning points, beginning actions, ending points, ending actions) *Let $Gr^p_{R_\alpha}$, be the precedence graph of the complex action α and let R_α be the selected recipe for executing α. Let $\beta_1, \ldots, \beta_n$ be the subactions of recipe R_α.*

1. *The set of vertices $\{s_{\beta_{b_1}}, \ldots, s_{\beta_{b_m}}\} \subseteq V^p_{R_\alpha}$ with in-degree 0 of the graph $Gr^p_{R_\alpha}$ are called* beginning points. *The actions $\beta_{b_1}, \ldots, \beta_{b_m}$ are called* beginning actions.

2. *The set of vertices $f_{\beta_{e_1}}, \ldots, f_{\beta_{e_m}} \subseteq V^p_{R_\alpha}$ with out-degree 0 of the graph $Gr^p_{R_\alpha}$ are called* ending points. *The actions $\beta_{e_1}, \ldots, \beta_{e_m}$ are called* ending actions.

[6] We will use the notation and will refer to a node which is labeled by s as node s.

This definition is illustrated in the following example:

Example 2 :
In the precedence graph of α $Gr^p_{R_\alpha}$ *in Figure 4, the vertex* s_{β_1} *is the only vertex with an in-degree of 0. Thus, only action* β_1 *is a beginning action and the time point* s_{β_1} *is a beginning point. Since vertex* s_{β_1} *has no predecessors the agent can start executing action* α *by executing subaction* β_1*. The vertices* f_{β_4} *and* f_{β_5}*, which have no successors, are called ending points. Thus, actions* β_4 *and* β_5 *are ending actions and the agent finishes executing action* α *when it finishes the execution of subactions* β_4 *and* β_5*.*

As a result of the dynamic nature of the planning process, the agent may have only partial knowledge of how to perform a complex level action. Thus, the agent may not know the duration of the execution time which is required for performing a complex level action until its plan for this action becomes complete. However, we assume that the time period during which a basic action is executed is always known. Thus, in our reasoning mechanism, we distinguish between complex and basic actions in the graph, as described in the following definition.

Definition 4.5 (basic edge, basic vertex, complex edge, complex vertex) :
Let $Gr^p_{R_\alpha}=(V^p_{R_\alpha},E^p_{R_\alpha})$ *be the precedence graph of an action* α*. Let* $(s_{\beta_i}, f_{\beta_i})$ *be some edge in* $E^p_{R_\alpha}$ *which denotes relations between the start time point,* s_{β_i}*, and the finish time point,* f_{β_i}*, of subaction* β_i*.*

1. *If subaction* β_i *is a basic action, the edge* $(s_{\beta_i}, f_{\beta_i})$ *is called a* basic edge *and the vertices* s_{β_i} *and* f_{β_i}*, are called* basic vertices.
2. *If subaction* β_i *is a complex action, the edge* $(s_{\beta_i}, f_{\beta_i})$ *is called a* complex edge *and the vertices* s_{β_i} *and* f_{β_i}*, are called* complex vertices.

Examples of such actions and edges are as follows:

Example 3 :
Suppose that subactions β_2 *and* β_5 *of* R^1_α *in Figure 3 are basic level actions and that the other subactions; i.e.,* β_1*,* β_3 *and* β_4*, are complex actions. Thus, the edges* $(s_{\beta_2}, f_{\beta_2})$ *and* $(s_{\beta_5}, f_{\beta_5})$ *in Figure 4 are called basic edges. The time points* s_{β_2}*,* f_{β_2}*,* s_{β_5}*,* f_{β_5} *are called basic vertices. The other edges are called complex edges, and the other vertices are called complex vertices.*

One of the requirements of our system is the ability to deal with metric information. We consider time points as the variables we wish to constrain. A time point may be a start or a finish point of some action α, as well as a neutral point of time such as 4:00p.m. Malik and Binford [38] have suggested constraining the *temporal distance* between time points. Namely, if X_i and X_j are two time points, a constraint on their temporal distance would be of the form $X_j - X_i \leq c$, which gives rise to a set of linear inequalities on the X_i's. In the following we give the formal definition of the metric constraints in our system.

Definition 4.6 (Metric constraints) *Let $\{\alpha, \beta_1, \cdots, \beta_n\}$ be the actions that agent G has to perform, where α is the highest level action in the recipe tree which consists of these actions. Let $V = \{s_\alpha, s_{\beta_1}, \cdots, s_{\beta_n}, f_\alpha, f_{\beta_1}, \cdots, f_{\beta_n}\} \cup \{s_{\alpha_{plan}}\}$ be a set of variables where s_y and f_y represent the start and finish time points of some action $y \in \{\alpha, \beta_1, \cdots, \beta_n\}$ respectively, and the variable $s_{\alpha_{plan}}$ represents the time point when an agent G starts to plan action α. Let $v_i, v_j \in V$ be two different variables. The* metric constraints *between these variables consists of a set of inequalities on their differences, namely $\theta^m_\alpha = \{v_i, v_j (1 \le i, j \le |V|) | a_{i,j} \le (v_i - v_j) \le b_{i,j}\}$.*

An example of metric constraints is described in Example 4.

Example 4 :

The recipe in Figure 3 consists of the following metric constraints:
(finish-time T_2- start-time $T_1 \le 20 minutes$),
(start-time T_4- finish-time $T_2 \le 40 minutes$),
(finish-time T_5- start-time $T_3 \le 60 minutes$),
(start-time T_3 after 5:00),
where T_α represents the time interval of the execution of action α. If we assume that the agent starts to plan α at 4:00 o'clock, then the agent may transfer these temporal constraints to the following metric constraints[7]*: $\theta^m_\alpha = \{0 \le f_{\beta_2} - s_{\beta_1} \le 20, 0 \le s_{\beta_4} - f_{\beta_2} \le 40, 0 \le f_{\beta_5} - s_{\beta_3} \le 60, 60 \le s_{\beta_3} \le \infty\}$.*

As mentioned in section 4, we shall use constraints networks in our system. These networks are frequently used in AI to represent sets of values which may be assigned to variables. In order to build a temporal constraints network, we have to build the temporal constraints graph which will consist of all the temporal constraints that are associated with an action α (including precedence constraints and metric constraints). However, because the environment of the agents is uncertain and dynamic (i.e., the agent may have only partial knowledge of how it will perform action α), the temporal network which we build will consist of only partial knowledge. When this knowledge becomes more complete, the agent will update the temporal network accordingly. That is, the agent builds the temporal network dynamically. At the beginning, when the agent adopts the intention of performing a high level action, α, it builds an initial temporal network with the initial information which it has on α, and it expands this initial network dynamically. The graph which consists of the initial constraints associated with action α is called, in our terminology, the initial graph of α, $InitGr_\alpha$, as described in the following definition.

Definition 4.7 (Initial graph of α, $InitGr_\alpha$) *Suppose that agent G has the intention to do an action α. Let α be the highest level action in the recipe tree and $\theta^m_\alpha = \{v_i, v_j | a_{i,j} \le (v_i - v_j) \le b_{i,j}\}$ be the temporal metric constraints associated with α. The* initial graph *of α, $InitGr_\alpha = (V_{init}, E_{init})$, satisfies the following.*

[7] The method of transferring metric information, which is given as natural points for a set of inequalities, can be found in several references, including Dechter et al. [14].

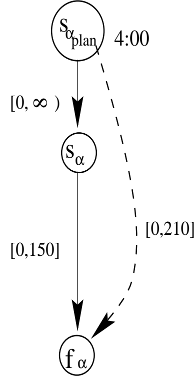

Figure 5: Example $InitGr_\alpha$.

1. *$V_{init} = \{s_{\alpha_{plan}}, s_\alpha, f_\alpha\}$ is a set of vertices. The vertex $s_{\alpha_{plan}} \in V_{init}$ denotes the time point at which the agent starts to plan action α. This point is a fixed point in time which is called the* origin. *The vertex s_α is the time point at which the agent starts executing α, and f_α is the time point at which the agent finishes executing α.*

2. *$E_{init} = \{(s_{\alpha_{plan}}, s_\alpha), (s_\alpha, f_\alpha), (s_{\alpha_{plan}}, f_\alpha)\}$ is a set of edges, where each edge $(v_i, v_j) \in E_{init}$ is labeled by the following weight:*

$$weight(v_i, v_j) = \begin{cases} [a_{i,j}, b_{i,j}] & \text{if } v_i, v_j \in \theta_\alpha^m \\ \\ [0, \infty) & \text{if } v_i, v_j \notin \theta_\alpha^m \end{cases}$$

.

An example of $InitGr_\alpha$ for some action α is given next.

Example 5 :
Suppose that α is associated with the following temporal constraints: (a) the performance of α has to be terminated in 150 minutes; (b) the performance of α has to start after 4:00 o'clock; (c) the performance of α has to end before 7:30 o'clock. We also assume, as in example 4, that the agent starts to plan α at 4:00 o'clock. Thus, the agent may transfer these temporal constraints to the following metric constraints, $\theta_\alpha^m = \{(0 \leq s_\alpha - f_\alpha \leq 150), (0 \leq (f_\alpha - s_{\alpha_{paln}} \leq 210)\}$, and to build the initial graph of action α, which is given in Figure 5.

When working on action α, the agent maintains a temporal constraint graph of α. This graph may be changed during the agent's planning. The following definition presents a formal description of this temporal constraint graph.

Definition 4.8 (Temporal constrains graph of α, Gr_α) *Let $\{\alpha, \beta_1, \cdots, \beta_n\}$ be a set of basic and complex level actions that agent G intends to perform, where α is the highest level action in the recipe tree which consists of these actions. Let $V = \{s_\alpha, s_{\beta_1}, \cdots, s_{\beta_n}, f_\alpha, f_{\beta_1}, \cdots, f_{\beta_n}\} \cup$*

$\{s_{\alpha_{plan}}\}$ *be a set of variables, where* s_y *and* f_y *represent the start and finish time points of some action* $y \in \{\alpha, \beta_1, \cdots, \beta_n\}$*, respectively, and the variable* $s_{\alpha_{plan}}$ *represents the time point that an agent* G *starts to plan action* α*. Let* $\theta^p_{R_\alpha} = \{(\beta_i, \beta_j) | \beta_i < \beta_j; i \neq j\}$ *be the set of all the precedence constraints which are associated with all the recipes of the actions* $\{\alpha, \beta_1, \cdots, \beta_n\}$*. Let* $\theta^m_\alpha = \{v_i, v_j | a_{i,j} \leq (v_i - v_j) \leq b_{i,j}\}$ *be all of the temporal metric constraints that are associated with* $\{\alpha, \beta_1, \cdots, \beta_n\}$ *and their recipes.*

1. *If* α *is a basic level action, then the temporal constraints graph of* α*,* $Gr_\alpha = (V_\alpha, E_\alpha)$*, is the initial graph* $InitGr_\alpha = (V_{init}, E_{init})$*.*

2. *If* α *is a complex level action, then the temporal constrains graph of* α*,* $Gr_\alpha = (V_\alpha, E_\alpha)$*, satisfies the following:*

   (a) *There is a vertex which represents the* **origin time point** $s_{\alpha_{plan}} \in V_\alpha$*.*

   (b) *There is a set of* **basic vertices** $V_{basic} = \{s_{\beta b_1} \cdots s_{\beta b_k}, f_{\beta b_1}, \ldots, f_{\beta b_k}\} \subseteq V_\alpha$ *and a set of* **basic edges** $E_{basic} = \{(s_{\beta b_1}, f_{\beta b_1}), \ldots, (s_{\beta b_k}, f_{\beta b_k})\} \subseteq E_\alpha$*, where* β_{b_i} $(1 \leq i \leq k)$ *is some basic level action in the recipe tree of* α*, and each edge* $(s_{\beta b_i}, f_{\beta b_i})\} \in E_\alpha$ *is labeled by the time period of* β_{b_i}*.*

   (c) *There is a set of* **complex vertices** $V_{comlex} = \{s_{\beta c_1}, \ldots, s_{\beta c_l}, f_{\beta c_1}, \ldots, f_{\beta c_l}\} \subseteq V_\alpha$ *and a set of* **complex edges** $E_{complex} = \{(s_{\beta c_1}, f_{\beta c_1}), \ldots, (s_{\beta c_l}, f_{\beta c_l})\} \subseteq E_\alpha$ *where each edge* $(s_{\beta c_j}, f_{\beta c_j}) \in E_{complex}$*,* $1 \leq j \leq l$ *is labeled by the following weight:*

$$weight(s_{\beta c_j}, f_{\beta c_j}) = \begin{cases} [a_{i,j}, b_{i,j}] & \text{if } (s_{\beta c_j}, f_{\beta c_j} \in \theta^m_\alpha) \text{and the time period of } \beta c_j \text{ is unknown} \\ \text{the time period of } \beta c_j & \text{if } (s_{\beta c_j}, f_{\beta c_j} \notin \theta^m_\alpha) \text{and the time period of } \beta c_j \text{ is known} \\ [a_{i,j}, b_{i,j}] \cap (\text{the time period of } \beta c_j) & \text{if } (s_{\beta c_j}, f_{\beta c_j} \in \theta^m_\alpha) \text{and the time period of } \beta c_j \text{ is known} \\ [0, \infty) & \text{if } (s_{\beta c_j}, f_{\beta c_j} \notin \theta^m_\alpha) \text{and the time period of } \beta c_j \text{ is unknown} \end{cases}$$

   (d) *There is a set of* **delay edges** $E_{delay} = \{(u_1, v_1), \ldots, (u_n, v_n)\} \subseteq E_\alpha$*, where* $u_i, v_i \in V_\alpha$*,* $(1 \leq i \leq n)$*. The vertices* u_i, v_i *may be of the following forms:*

      i. u_i *is the start time point of some action* β_u*, and* v_i *is some beginning point in the precedence graph* $Gr^{time_p}_{\beta_u}$ *of action* β_u*.*

      ii. u_i *is some ending point in the precedence graph* $Gr^{time_p}_{\beta_v}$ *of some action* β_v*, and* v_i *is the finish time point of action* β_v*.*

      iii. u_i *is the finish time point of some action* β_u*, and* v_i *is a start time point of another action* β_v*, where* $\beta_u, \beta_v \in \theta^{time_p}_{R_\alpha}$*.*

   Each edge $(v_i, v_j) \in E_{delay}$ *is labeled by the following weight:*

$$weight(v_i, v_j) = \begin{cases} [a_{i,j}, b_{i,j}] & if\ v_i, v_j \in \theta_\alpha^m \\ [0, \infty) & if\ v_i, v_j \notin \theta_\alpha^m \end{cases}$$

(e) *There is a set of directed edges* $E_{metric} = \{(u_1, v_1), \ldots, (u_n, v_n)\} \subseteq E_\alpha$ *labeled by its metric constraints, where* $u_i, v_i \in V_\alpha$, $(1 \leq i \leq n)$, *but* $(u_i, vi) \notin E_{basic} \cup E_{complex} \cup E_{delay}$.

Example 6 :
Figure 6 illustrates an example of a temporal constraints graph of α. *Action* α *is a complex level action, and thus the vertices* s_α *and* f_α *are complex vertices. The edge* (s_α, f_α) *is a complex edge. The vertex* $s_{\alpha_{plan}}$ *is the origin time point which represents the time 4:00 o'clock, at which time the agent has begun its plan for* α. *The actions* β_1, β_2, β_3, β_4, *and* β_5 *are the subactions in the recipe which is selected for performing* α. *As a result, all of the vertices which represent the start and finish time points of these subactions appear in the path between* s_α *and* f_α. *Some of these subactions may be multi-agent actions, and the others are single agent actions. In this example we assume that* β_1 *is a single agent action and that the recipe which is selected for performing this action consists of subactions* γ_{11} *and* γ_{12}, *which are basic level actions. Thus, the vertices* $\{s_{\gamma_{11}}, f_{\gamma_{11}}, s_{\gamma_{12}}, f_{\gamma_{12}}\}$ *are called in our terminology basic vertices, and the edges* $(s_{\gamma_{11}}, f_{\gamma_{11}})$, $(s_{\gamma_{12}}, f_{\gamma_{12}})$ *are called basic edges. Note that the time period during which a basic action is executed is always known. As a result, the weights of these basic edges are fixed (exactly 5 minutes). However, since the duration of complex level actions is not known in advance and may change over time, the weights of the complex edges in the graph are not fixed. The next section describes the building of this graph.*

As mentioned above, some of the vertices in the temporal constraints graph may be fixed; i.e., these vertices denote a known time which cannot be modified. The vertex $s_{\alpha_{plan}}$ represents the time point at which an agent G starts to plan action α; it is a fixed vertex. The value of this vertex in Figure 6 is 4:00 o'clock. Other fixed vertices may be initiated, for example, by a request from a collaborator. The vertex s_{β_5} is also a fixed vertex, which represents the time 5:00 o'clock. We define fixed time points as follows:

Definition 4.9 (fixed time point) *A* fixed time point *is a known time which cannot be modified.*

In section 3 we illustrated the recipe tree for action α. This recipe tree may be derived from the temporal constraints graph of α and in our terminology is called the implicit recipe tree. This tree is described in the following definition.

Definition 4.10 (Implicit recipe tree, $Tree_\alpha$:) *Let* Gr_α *be a constraints graph. Let* $\mathcal{A} = \{\alpha, \beta_1, \ldots, \beta_n\}$ *be the set of the actions whose start and finish time points are represented by the vertices in* Gr_α. *The implicit recipe tree,* $Tree_\alpha$, *in* Gr_α *satisfies the following: each node in the recipe tree represents an action from the set* $\mathcal{A}$, *where the root of this tree is action* α.

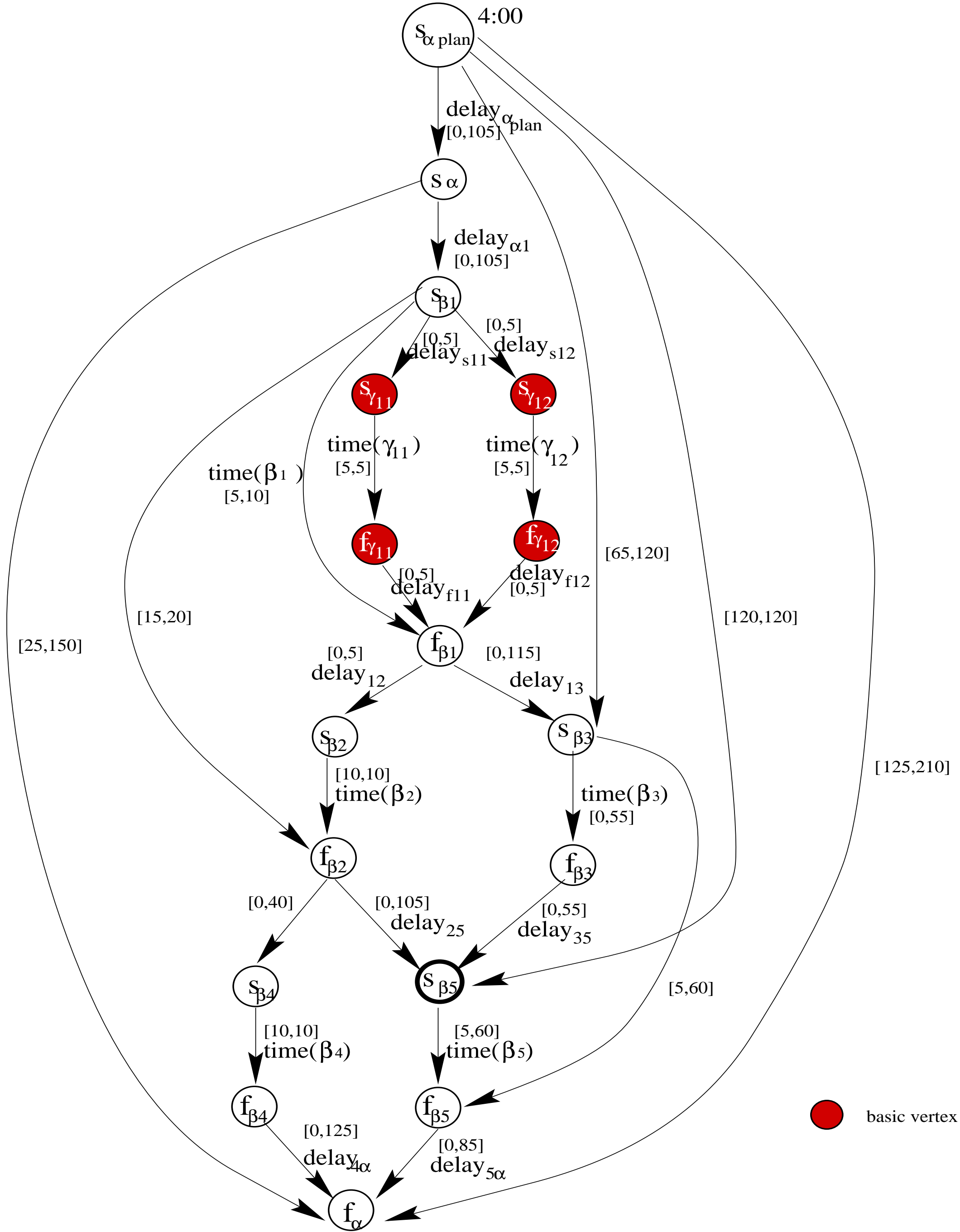

Figure 6: An example of a temporal constraints graph Gr_α.

There is an edge between a node $x \in \mathcal{A}$ to another node $y \in \mathcal{A}$ (i.e., y is a child of x in the tree) if Gr_α consists of the following vertices path $(s_x, \ldots s_y, f_y, \ldots f_x)$ and there is no other action $z \in \mathcal{A}$ that exists, such that Gr_α consists of the path $(s_x, \ldots, s_z \ldots, s_y, f_y, \ldots, f_z, \ldots, f_x)$.

Example 7 :
Figure 7 presents the implicit recipe tree of the graph Gr_α *which is shown in Figure 6. It is obvious that* $\mathcal{A} = \{\alpha, \beta_1, \cdots, \beta_5, \gamma_{11}, \gamma_{12}\}$. *In this tree, for example,* γ_{11} *is the son of* β_1 *since* Gr_α *consists of the path* $(s_{\beta_1}, s_{\gamma_{11}}, f_{\gamma_{11}}, f_{\beta_1})$.

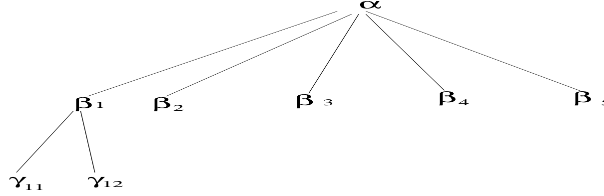

Figure 7: Implicit recipe tree example.

4.2 The Algorithm

Figure 8 presents the major constituents of the temporal reasoning algorithm which is used by the AI planning subsystem. The algorithm obtains as an input an action α and an initial graph $InitGr_\alpha$, that is defined in definition 5. As shown in Figure 8, the algorithm consists of two major constituents. In the first constituent, the agent initializes the relevant parameters. In particular, it forms the graph Gr_α. Then, in the second constitute it runs a loop until all the subactions of a tree for α are performed. Figure 9 describes these constituents in more detail. The initialization constituent includes an initialization of the temporal constraints graph Gr_α by the initial graph $InitGr_\alpha$. It also includes initialization of the status of the vertices in this graph. In addition, it initializes two flags, where the

(I) Initialization and formation of the temporal graph of α (Gr_α);
(II) Planning and executing Loop:
  (II.1) Planning for a chosen subaction β from Gr_α:
    (II.1.a) β is a basic level action;
    (II.1.b) β is a complex level action;
  (II.2) Obtaining messages on execution from the RT scheduling subsystem;
  (II.3) Backtracking, if needed;

Figure 8: The major constituents of the temporal reasoning mechanism for performing a complex action α.

(I) Initialization : Initialize the values of the relevant parameters, including the set of *Independent Unexplored* vertices and the value of the temporal constraints graph of α, Gr_α;

(II) Planning and execution Loop: Run the following loop until all the basic actions have been executed:

- **(II.1) Planning for a chosen action β :** If the plan for α has not been finished, choose some vertex, s_β, from the IU set which is associated with an action β.
  - **(II.1a) If β is a basic action** : identify its parameters, send them to the RT scheduling subsystem and start again from (II.1).
  - **(II.1b) If β is a complex action** :
    - **(II.1b1)** Select a recipe R_β for β;
    - **(II.1b2)** Construct the precedence graph $Gr^p_{R_\alpha}$ for β and add it to Gr_α.
    - **(II.1b3)** Add the metrics constraints of the recipe of β to Gr_α.
    - **(II.1b4)** Check consistency of the updated graph using Floyd-Warshall's all-pairs-shortest-paths algorithm.
    - **(II.1b5)** If not consistent then remove $Gr^p_{R_\alpha}$ and all the metric edges added in (II.1b3) from Gr_α and proceed to (II.1b1) to select a new recipe. If such a new recipe for β does not exist, then return failure in the current plan.
- **(II.2) Obtaining messages from the RT scheduling on execution:** Listen to the RT scheduling subsystem as long as the agent did not execute all the actions and update the graph according to the messages of the RT scheduling subsystem.
- **(II.3) Backtracking, if need:** If there is a failure in the current plan or a failure message from the RT scheduling subsystem, then perform backtracking.

Figure 9: The description of the major constituents of the temporal reasoning algorithm.

first indicates whether or not the planning of α has terminated, and the second indicates whether or not all the subactions of α have already been executed. It also initializes the set of *Independent Unexplored* vertices that we discuss below.

The second constituent is a loop which runs until the agent has finished the execution of all the basic level actions in the complete recipe tree for α. This loop consists of an internal loop (II.1) which is run until the agent has finished the planning of action α. The two nested loops are needed since in several cases the agent fails in the execution of some basic level actions after the planning of α has been completed (i.e., after it finished building the complete recipe tree of α). As a result, the agent has to backtrack, and it tries to find a new plan for α by running the planning loop again.

In the planning loop an action β is chosen (the method of choosing this action will be discussed below). The agent distinguishes between a basic level action β (II.1.a) and a complex level action (II.1.b). If the action is a basic level action, it sends this action and its

associated time constraints to the RT scheduling subsystem, which schedules and executes this action. If the action is a complex level action, the agent does not know the duration of β. Thus, the agent continues to plan the performance of this action by selecting a recipe for this action. It also expands the temporal constraints graph Gr_α according to the new information. In the case of a failure in finding a recipe in the planning loop for a chosen β or in the case of receipt of a failure message from the RT scheduling subsystem, the agent performs backtracking, as described in section 4.3.

The pseudo code of the reasoning algorithm's main procedure is given in Figure 10. In each line of the pseudo code we indicated the beginning of the appropriate constituent from Figure 9. We demonstrate the algorithm using an example because it is complex.

To keep track of the progress of the expansion of Gr_α, all the vertices start out as *unexplored* (UE). When they are explored during the algorithm, the vertices of basic actions become *explored basic* (EB), and the vertices of complex actions become *explored complex* (EC). The algorithm also takes into consideration that if the performance of some action β depends upon other actions $\{\gamma_1, \ldots, \gamma_n\}$, then the agent cannot determine the time needed to perform β unless the finish time of $\{\gamma_1, \ldots, \gamma_n\}$ is known. A vertex whose time can be determined and which does not depend on other actions is called an *Independent Vertex (IndV)*, and it belongs to one of two disjoint sets. The first set, IU, contains the *Independent Unexplored* vertices, i.e., the vertices whose values can be identified, but have not yet been handled by the algorithm. The second set, IE, contains the *Independent Explored* vertices, i.e., the vertices whose values have been identified. Table 2 provides a summary of the notation used in the temporal reasoning of this paper.

To exemplify the running of the algorithm in Figure 10, we suppose that the agent has adopted an intention of performing an action α which is associated with the metric time constraints given in example 5: i.e., $\theta^m_\alpha = \{(0 \leq s_\alpha - f_\alpha \leq 150), (0 \leq (f_\alpha - s_{\alpha_{paln}} \leq 210)\}$. We also assume that, initially, the length of the interval of the execution of action α is unknown. In order to start the temporal reasoning process, before applying the algorithm IDENTIFY_TIMES_PARAM, the agent first builds an initial temporal graph (see definition 4.7) denoted by $InitGr_\alpha = (V_{init}, E_{init})$, which includes the initial information of the time constraints of the highest level action α. In the example, if we assume that the agent starts its plan at 4pm, then $V_{init} = \{s_{\alpha_{plan}}, s_\alpha, f_\alpha\}$ and $E_{init} = \{(s_{\alpha_{plan}}, s_\alpha, [0, \infty]), (s_\alpha, f_\alpha, [0, 150]), (s_{\alpha_{plan}}, f_\alpha, [0, 210])\}$ (see Figure 5). The initial graph, $InitGr_\alpha = (V_{init}, E_{init})$, is built according to definition 4.7 and is given as an input to the procedure IDENTIFY_TIMES_PARAM.

In the initialization constituent (clause (I)) of the algorithm, the agent first checks if the initial graph is consistent (line **1** of the main procedure of Figure 10). For simplicity, we assume that each edge in the graph is labeled by a single interval. As a result, we can apply Floyd-Warshall's all-pairs-shortest-paths algorithm [10] in the CHECK_CONSISTENCY procedure[8]. After applying the Floyd-Warshall's algorithm, the new intervals of the edges in the example will be $E_{init} = \{(s_{\alpha_{plan}}, s_\alpha, [0, 210]), (s_\alpha, f_\alpha, [0, 150]), (s_{\alpha_{plan}}, f_\alpha, [0, 210])\}$ (see the dashed edges in Figure 12).

If the initial graph is consistent, as in our example, $InitGr_\alpha$ is assigned to Gr_α (line

[8]The general problem of edges being labeled by several intervals can be solved using heuristic algorithms, as suggested in [14].

IDENTIFY_TIMES_PARAM$(\alpha, InitGr_\alpha)$

(I) **1** **if** CHECK_CONSISTENCY$(InitGr_\alpha)$ is false **then FAIL**;
2 $Gr_{\alpha_{times}} \leftarrow InitGr_\alpha$; IE $\leftarrow \emptyset$;
3 IU $\leftarrow$ (the fixed vertices in Gr_α) ;
4 **for each** vertex $u \in V_\alpha$ **do** status$[u] \leftarrow$ UE;
5 status$[s_{\alpha_{plan}}] \leftarrow$ EC;
6 finish_execute_all $\leftarrow false$; complete_plan $\leftarrow false$;
7 UPDATE_INDV_SETS$(s_{\alpha_{plan}})$;
(II) **8** **while**(not finish_execute_all)
(II.1) **9** **if** (not complete_plan) **then**
10 select some vertex u from IU;
11 $\beta \leftarrow$ action$[u]$;
(II.1a) **12** **if** β is a basic action **then**:
13 status$[s_\beta] \leftarrow$ EB; status$[f_\beta] \leftarrow$ EB;
14 UPDATE_INDV_SETS(s_β, f_β)
15 $D_\beta \leftarrow time_period(\beta)$;
16 $r_\beta \leftarrow time(s_{\alpha_{plan}}) + M(s_{\alpha_{plan}}, s_\beta)$;
17 $d_\beta \leftarrow time(s_{\alpha_{plan}}) + M(s_{\alpha_{plan}}, f_\beta)$;
18 $p_\beta \leftarrow$ FIND_PRECEDENCE_ACTIONS(β, Gr_α);
19 transfer $\langle \beta, D_\beta, r_\beta, d_\beta, p_\beta \rangle$ to RT scheduling subsystem
which tries schedule β and informs "succeed" or "fail"
(II.1b) **20** **if** β is a complex action **then:**
21 found $\leftarrow$ false; $\Re_{\beta_{tried}} \leftarrow \emptyset$;
22 **while** (not found)$\wedge(\Re_\beta - \Re_{\beta_{tried}} \neq \emptyset)$
(II.1b1) **23** Select_Rec R_β for β not in $R_{\beta_{tried}}$;
24 add R_β to $\Re_{\beta_{tried}}$;
(II.1b2) **25** $Gr^p_{R_\alpha} \leftarrow$ CONSTRUCT_PRECEDENCE_GRAPH(R_β);
26 ADD_PRECEDENCE_GRAPH$((s_\beta, f_\beta), Gr^p_{R_\alpha})$;
(II.1b3) **27** ADD_METRIC_CONSTRAINTS(R_β);
(II.1b4) **28** **if** CHECK_CONSISTENCY(Gr_α) **then**:
29 status$[s_\beta] \leftarrow$ EC; status$[f_\beta] \leftarrow$ EC;
30 found $\leftarrow$ true;
(II.1b5) **31** UPDATE_INDV_SETS(s_β);
32 **else**, Gr_α is not consistent, REMOVE(R_β);
33 **end while**;
34 **if** all the vertices in the graph are explored **then**:
35 complete_plan $\leftarrow true$;
36 **end if** (not complete_plan);
(II.2) **37** message $\leftarrow$ LISTEN_TO_THE_REAL-TIME_SUBSYSTEM;
38 **if** the message is "succeed to execute an action β" **then**
39 status$[s_\gamma] \leftarrow$ EXECUTED; status$[f_\gamma] \leftarrow$EXECUTED;
40 **if** the status of all the vertices in the graph are EC or EXECUTED
41 **then** finish_execute_all $\leftarrow true$;
(II.3) **42** **if** (not found) or (RT scheduling subsystem sent "fail")
then: if possible, **BACKTRACK**; otherwise **FAIL**;
43 **end while**;

Figure 10: The main procedure for identifying the time parameters of α where α is a single agent action.

2), and the expansion of the graph continues. At the beginning, the first vertex $s_{\alpha_{plan}}$ is denoted as EC (line **7**). Then, the IU and IE sets are updated by the procedure UP-

Notation	Meaning	Comments
α	action	Also: β, β_i
R_α	recipe for α	
$\Re_\beta$	set of recipes for β	Also: $\Re_{\beta_{tried}}$
s_α	time point at which α starts	definition 4.1
f_α	time point at which α ends	definition 4.1
G	agent	
GR	group	
$Gr^p_{R_\alpha} = (V^p_{R_\alpha}, E^p_{R_\alpha})$	precedence graph of α	definition 4.2
$\theta^p_{R_\alpha}$	precedence constraints in R_α	
$\theta^m_{R_\alpha}$	temporal metric constraints in R_α	
$InitGr_\alpha = (V_{init}, E_{init})$	Initial graph of α	definition 4.7
$Gr_\alpha = (V_\alpha, E_\alpha)$	temporal constraints graph of α	definition 4.8
θ^m_α	temporal metric constraints of α	
θ_α	temporal constraints of α and α's constitutes	
D_β	duration time of β	
r_β	release time of β	
d_β	deadline of β	
p_β	predecessor actions of β	
UE	unexplored vertices	
EB	explored basic vertices	
EC	explored complex vertices	
IndV	independent vertex	
IE	explored independent vertex	
IU	unexplored independent vertex	

Table 2: Summary of notation used for special variables and constants.

DATE_INDV_SETS($s_{\alpha_{plan}}$). The procedure UPDATE_INDV_SETS (applied in lines **9, 15** and **33**) is described in Figure 11. Suppose that UPDATE_INDV_SETS procedure receives as an input a vertex u whose time (if it is a basic vertex) or a preliminary plan (if it is a complex vertex) has just been identified. First UPDATE_INDV_SETS algorithm adds the vertex u to the IU set. Then, for each adjacent vertex v of u (i.e., $(u, v) \in E$), it checks if all the vertices which enter v (i.e., for each vertex a such $(a, v) \in E$) belong to IE. If so, it checks if the status of v is an unexplored vertex (UE). If so, it adds v to IU. Otherwise (if the status of v is either explored basic (EB) or explored complex (EC)), it runs the procedure UPDATE_INDV_SETS(v) recursively on v. (We prove, in Appendix A, that if Gr_α consists of some UE vertices, then at least one of these vertices is an IndV.) Thus, at each stage of the algorithm, the agent selects a UE vertex from IU and identifies the time value of this vertex. After applying UPDATE_INDV_SETS($s_{\alpha_{plan}}$) (line **9**), IU contains s_α that is adjacent to the unique vertex, which is a fixed vertex. Thus, the agent starts its execution and planning loop (clause (II)) by developing the temporal constraints graph of α from s_α.

In the first part of the loop (clause (II.1)) of developing the graph, the agent distinguishes between vertices associated with basic actions (clause (II.1a)-lines **13-20**) and vertices associated with complex actions (clause (II.1b)-lines **21-35**). In the example given, the vertex s_α

```
UPDATE_INDV_SETS(u)
1  IE ← IE ∪ u; IU ← IU − u;
2  for each vertex (u, v) ∈ E_α do
3     BecomesIV ← True;
4     for each vertex (a, v) ∈ E_α do
5       if a ∉ IE then BecomesIV ← False;
6     if BecomesIV = True then
7       if status[v] = UE then IU ← IU ∪ v;
8         otherwise: UPDATE_IV_SETS(v);
```

Figure 11: The procedure UPDATE_IV_SETS updates the sets of the Independent Vertices.

denotes a time point of a complex action. Thus, the agent selects some applicable recipe R_α for this action (clause (II.1b1)-line **25**). Then, the agent builds the precedence graph of R_α, $Gr^p_{R_\alpha}$, by running the CONSTRUCT_PRECEDENCE_GRAPH procedure (clause (II.1b2)-line **27**). $Gr^p_{R_\alpha}$ is built according to definition 4.2. In our example, we assume that the agent selects a recipe R^1_α which is associated with the subactions $\{\beta_1, \beta_2, \beta_3\}$ and with precedence constraints $\{\beta_1 < \beta_2,\ \beta_1 < \beta_3\}$ in which β_1 is a basic level action. Thus, $V^p = \{s_{\beta_1}, \ldots, s_{\beta_3}, f_{\beta_1}, \ldots, f_{\beta_3}\}$ $E^p = \{(s_{\beta_1}, f_{\beta_1}), \ldots, (s_{\beta_3}, f_{\beta_3}), (f_{\beta_1}, s_{\beta_2}), (f_{\beta_1}, s_{\beta_3})\}$.

After building $Gr^p_{R_\alpha}$, it is incorporated into Gr_α by adding directed edges from s_α to the beginning points of $Gr^p_{R_\alpha}$ and adding edges from the ending points of $Gr^p_{R_\alpha}$ to f_α. That is, it adds directed edges from s_α to each vertex $v_{b_1}, \ldots, v_{b_m} \subseteq V^p$ with an indegree of 0 and adds edges from each vertex $v_{e_1}, \ldots v_{e_k} \subseteq V^p$ with an outdegree of 0 to f_α (line **28**). A pictorial illustration of running the procedure ADD_RECIPE_SUBACTIONS on the example is given in Figure 12. The precedence graph is incorporated into $Gr_{\alpha_{times}}$ by adding the edges $(s_\alpha, s_{\beta_1}), (f_{\beta_2}, f_\alpha), (f_{\beta_3}, f_\alpha)$. After this, the agent also updates the metric constraints which are associated with the appropriate recipe (Figure 13(A)). In the example, the metric constraints given in R^1_α are: $\{(s_{\beta_3} - f_{\beta_1} \leq 60)$, ($s_{\beta_3}$ after 5:00) }.

When the agent continues with its planning for α, it has to ensure that all the time constraints are satisfied (clause (II.1b4) - line **28**). Thus, if the graph is inconsistent, it returns to the previous graph which was consistent by removing the new edges and vertices that have been added to the graph when the last R_β was chosen (clause (II.1b5) - line **34**). Then, it tries to select another recipe for β (line **25**), until it finds a consistent recipe or fails.

If there is a failure at any stage of the planning, it is very easy to remove the edges and the vertices of a specific recipe of a certain action β since, according to the construction of the graph, all of them are between s_β and f_β. It is also easy to add new constraints required by changes in the environment by adding appropriate edges. In both cases, the Floyd-Warshall's algorithm is used to update the weights. Thus, in the case of failure to find a consistent recipe in the planning loop for a chosen β (i.e., in clause (II)), the agent removes the appropriate edges and vertices from G_α and changes the status of s_β and f_β back to UE. The agent's backtracking mechanism is discussed in section 4.3.

In the example given, the graph is consistent; thus the agent continues to develop the graph Gr_α by selecting another unexplored (UE) independent vertex from the IU set. Suppose that the vertex s_{β_2} is the fixed time point 6pm, the performance time of the basic action

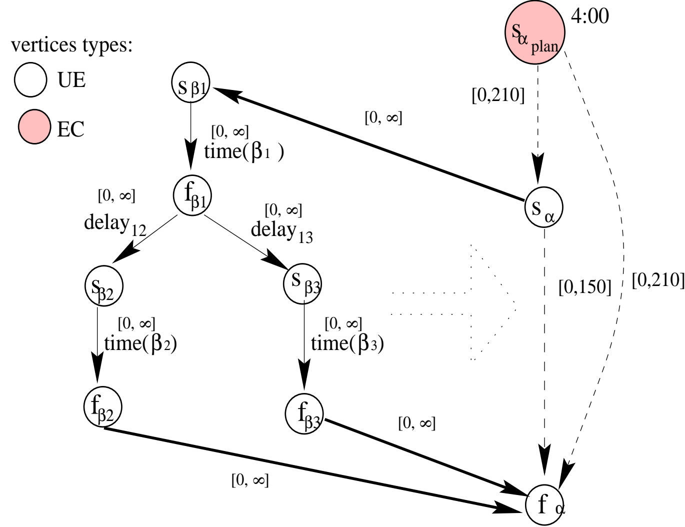

Figure 12: Example of running the procedure ADD_RECIPE_SUBACTIONS on $InitGr_{\alpha}$.

β_1 is exactly 10 minutes, and β_2 may take between 5 to 235 minutes. After applying the procedure UPDATE_INDV_SETS(s_{α}), the IU set contains s_{β_1} and s_{β_2}.

If the procedure selects a basic action from IU (clause (II.1a) line **13**), it should identify $\langle D_{\beta}, r_{\beta}, d_{\beta}, p_{\beta}\rangle$. D_{β} is derived from the entities of the basic action (we assume that the performance time of each basic action is known). d_{β} and r_{β} are derived from the distance graph, which is returned by the Floyd-Warshall algorithm and is maintained in a matrix M. p_{β} is identified by the FIND_PRECEDENCE_ACTION procedure, which finds all the EB vertices that are ancestors of β and adds their associated actions to p_{β}. β and these values are sent to the RT scheduling subsystem, and the decision regarding the exact time β will be executed is left to the RT scheduling subsystem. In the example, if the basic action β_1 is selected, then the status of s_{β_1} and f_{β_1} becomes EB (Figure 13), $D_{\beta_1} = 10$, $r_{\beta_1} =$ 4pm, $d_{\beta_1} =$ 6pm, $p_{\beta_1} = \emptyset$.

4.3 The Backtracking Method

As we have mentioned, our algorithm enables the agents to backtrack and to change their timetables easily. In the current system, the agent backtracks in two cases. The first case is when the agent is unable to continue its plan for performing α under the appropriate time constraints of α. As a result, the agent has to backtrack in order to find another plan for performing α. The second case is when the RT scheduling subsystem is unable to schedule various basics actions under their identified constraints. As a result, these actions are sent back to the AI planning subsystem, which tries to find a new way for planning and performing α. In both cases, the agent runs the backtracking mechanism which is described in Figure 14. In this mechanism we assume that the agent maintains the implicit recipe tree, $Tree_{\alpha}$, which is defined in Definition 4.10.

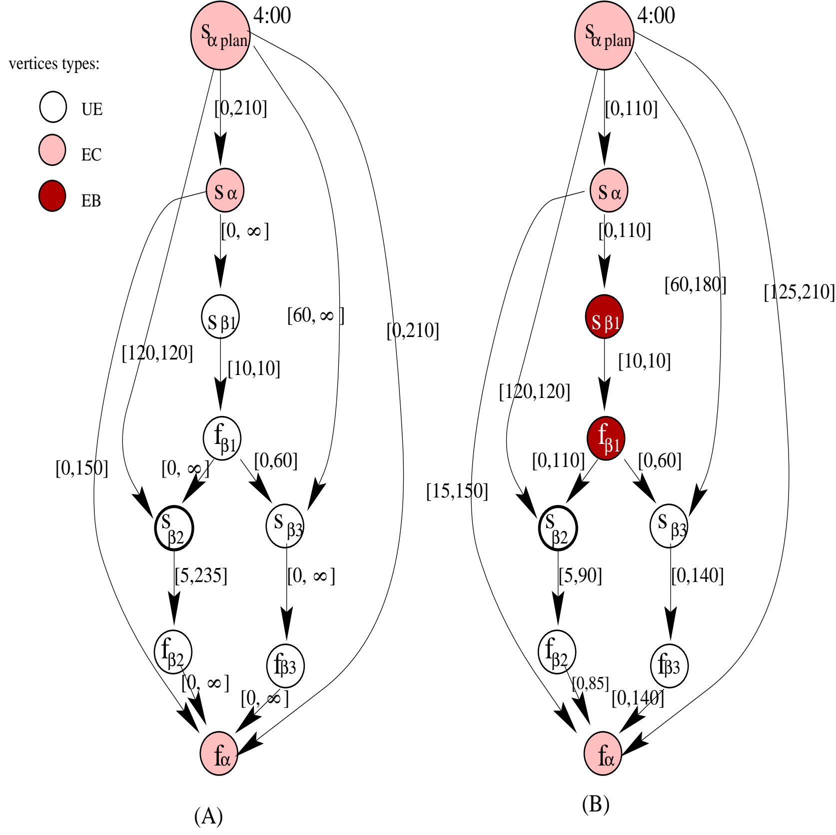

Figure 13: Examples of Gr_α: (A) after applying the ADD_METRIC_CONSTRAINTS procedure with the metric constrains $\{(s_{\beta_3} - f_{\beta_1} \leq 60)$, ($s_{\beta_3}$ after 5:00) $\}$ on the graph in Figure 12; (B) after applying the CHECK_CONSISTENCY procedure and updating status[s_β] and status[f_β].

Suppose that the BACKTRACK procedure (in Figure 14) receives action γ, a problematic action, as input. Also, we suppose that β is the father of γ in the implicit recipe tree of α, $Tree_\alpha$, which is developed until then. The agent selects a random node n_{curr} from $Tree_\alpha$. Then it compares the number of its descendants which are not leaves in $Tree_\alpha$ to the number of descendants which are not leaves of β. It removes all of the descendants of the node with the lower number of descendants. This method is based on the random hill climbing algorithm, which is proved as a good heuristic in our environment.

After removing the descendants of the selected action (*selected_action*) from $Tree_\alpha$, it removes all of the relevant information from Gr_α by running the REMOVE procedure which is described in Figure 24 in Appendix B. As shown in this figure, it is very easy to remove the relevant information from the graph. The procedure just has to delete the vertices which indicate the start and finish time points of the descendants of the *selected_action*. It also removes these vertices from the IE and IU sets. Finally, it removes all the edges which are connected to these vertices. Then it updates the *independent vertices* sets by using the procedure UPDATE_IV_SETS (see Figure 11). After the changing of $Tree_\alpha$ and Gr_α it tries

BACKTRACK $(\gamma, Tree_\alpha)$

1 $\beta \leftarrow$ the father of γ in $Tree_\alpha$;
2 $ComplexSubact_\beta \leftarrow$ the number of the nodes which are descendants of β in $Tree_\alpha$ and are not leaves;
3 $complex_nodes_{set} \leftarrow$ all the actions in $Tree_\alpha$ which are not leaves;
4 $n_{curr} \leftarrow$ select randomize node from $complex_nodes_{set}$;
5 $complex_subact_{n_{curr}} \leftarrow$ the number of the nodes which are descendants of n_{curr} in $Tree_\alpha$ and are not leaves;
6 **If** $complex_subact_{n_{curr}} \geq complex_subact_\beta$
7 **then** $selected_action \leftarrow \beta$;
8 **else**, $selected_action \leftarrow$ the action which is represented by the node n_{curr};
9 $descend_{set} \leftarrow$ all the descendants of $selected_action$;
10 $Tree_{\alpha_{new}} \leftarrow$ cut all the descendants of $selected_action$ from $Tree_\alpha$;
11 REMOVE$(selected_action, descend_{set})$;
12 **If** $selected_action$ has additional recipe R_{new} which is consistent with performing α under the time constraints θ_α **then**
13 continue the planning for α by selecting R_{new};
14 **else,**
15 **If** the $selected_action$ represents action α (the root of $Tree_\alpha$) **then:**;
16 return **FAILURE**;
17 **else,** $\gamma' \leftarrow$ the action which is represented by the father of $selected_action$;
18 BACKTRACK $(\gamma', Tree_{\alpha_{new}})$;

Figure 14: Backtracking Mechanism.

to find a new plan by selecting a new consistent recipe for the *selected_action*. If such a recipe does not exist, it continues to backtrack by selecting additional new nodes. If the agent backtracks until the root of $Tree_\alpha$ without finding any consistent graph, it fails to find a plan for α.

5 The Correctness and the Complexity of the Algorithm

In this section we present the theorem which states the conditions for the correctness and completeness of the time reasoning algorithm in the AI planning subsystem which is described in the former section. The proof of this theorem is based on the lemmas and the propositions which are given in Appendix A.

Theorem 1 *Suppose that the agent needs a plan for some action α with a given set of constraints θ_α. (1) If the* IDENTIFY_TIMES_PARAM *terminates, such that the agent has identified a set of basic actions, $\{\beta_1, \ldots, \beta_k\}$, and corresponding time requirements (i.e., $\langle D_{\beta_i}, r_{\beta_i}, d_{\beta_i}, p_{\beta_i}\rangle$, $1 \leq i \leq k$), then, the performance of $\{\beta_1, \ldots, \beta_k\}$ (possibly in parallel), under all the identified constraints, constitutes performing α under θ_α. (2) Suppose that a complete recipe tree for α exists which satisfies the appropriate time constraints and for which the RT scheduling subsystem can find a feasible schedule consisting of the given basic actions*

and their time constraints. Then, the IDENTIFY_TIMES_PARAM *algorithm can identify the basic actions of such a tree and their constraints. If there is no complete recipe tree for* α *that satisfies the appropriate time constraints, or if the RT scheduling subsystem cannot find a feasible schedule, then the algorithm fails.*

Proof: (1) According to corollary 1 (in Appendix A), the algorithm terminates when the agent has identified a set of basic level actions, $\{\beta_1, \ldots, \beta_k\}$, and according to proposition 2 (in Appendix A), the agent can identify all the time parameters of these actions (i.e. $\langle C_{\beta_i}, r_{\beta_i}, d_{\beta_i}, p_{\beta_i} \rangle$, $1 \leq i \leq k$). Now we have to prove that performing $\{\beta_1, \ldots, \beta_k\}$ under the identified constraints constitutes performing α under constraints θ_α. But, according to corollary 1, when the agent finishes developing its plan for α, all the leaves in $Tree_\alpha$ are basic level actions. Thus, according to lemma 2 in Appendix A, performing the basic level actions, $\{\beta_1, \ldots, \beta_k\}$, under the identified constraints, constitutes performing α under constraints θ_α.

(2) Suppose that the agent did not find a recipe which constitutes performing α under constraints θ_α for some complex level action, while developing the implicit recipe tree for α. The agent can then use some known backtrack method, on the implicit recipe tree for α, which enables him to check all of the options of all the other appropriate available recipes. As a result, if such a recipe does not exist, the agent will fail in its plan. □

For the complexity analysis of the AI planning subsystem's algorithm, the number of nodes of the largest partial recipe tree that has been developed by the agent during the performance of the algorithm is denoted by m. k denotes the number of times the **Select_Rec** process was initiated by the algorithm. Since the complexity of the Floyd-Warshall algorithm is $O(n^3)$, the complexity of the algorithm is $O(km^3)$. If no backtracking is needed (e.g., there is one possible recipe for each action), then k is the number of complex actions in the complete recipe tree for α, and the algorithm is polynomial in the number of nodes of this tree. However, if backtracking is needed, k may be exponential in the size of the tree, and several heuristics can be used to limit the search.

6 The RT scheduling Subsystem

The RT scheduling subsystem is responsible for the scheduling and execution of the basic actions. The RT scheduling subsystem must schedule sets of actions generated by the AI planning subsystem. Therefore, the RT scheduling subsystem does not have complete knowledge about the action set and its constraints before starting the scheduling process. Consequently, the scheduler does not assume any knowledge of the characteristics of actions that have not yet been sent to the RT scheduling system; the schedule is computed on-line as actions arrive (during run time). The RT scheduling subsystem also includes a dispatcher responsible for executing the basic actions at the time determined by the scheduling process, even before the scheduling and the planning process have ended. This technique enables the agents to manage the times of critical tasks that have strict deadlines during the planning process. In addition, due to changes that occur in the dynamic environment, a feasible

```
BUILD_FEASIBLE_SCHEDULE()
 1  while(1)
 2    A_new ← get_basic_action_set from AI planning subsystem;
 3    if A_new then:
 4      t_c ← read_current_time();
 5      update the set A according to A_new;
 6      S_best ← BUILD_INITIAL_SCHEDULE(T_C);
 7      L_max(S_best) ← maximum lateness of S_best;
 8      if L_max(S_best) > 0 then:
 9        res ← IMPROVE_SCHEDULE(S_best);
10      if res=FAILURE then find the actions whose deadline
        were missed and send them back to the AI planning subsystem;
```

Figure 15: The pseudo code of the process in the RT scheduling subsystem which determines the schedule S_{best}. The variables $\mathcal{A}$ and S_{best} are global. For simplicity, we refer to these variables as sets. In addition, there is a dispatcher which executes the actions in S_{best} when their start time arrives. Then these actions are deleted from $\mathcal{A}$ and S_{best} by the dispatcher after it completes their execution.

schedule must sometimes be altered after having been previously determined. For example, agents may determine that the course of action they have committed to is not working. Therefore, a new course of action must be generated by the AI planning subsystem and a new scheduling process must be engaged by the RT scheduling subsystem. The RT scheduling subsystem must be able to find a new feasible schedule in response to the new changes. However, because our system interleaves planning and execution, some actions belonging to the former schedule may have already been executed by the RT scheduling subsystem. Thus, the former schedule may influence the agents' new planning. As a result, the RT scheduling subsystem must send update messages to the AI planning subsystem and vice versa. In this section we describe the scheduling algorithm which is used by the RT scheduling subsystem.

The goal of the scheduling algorithm is to find a feasible schedule for the basic actions set $\{\beta_i|(1 \leq i \leq n)\}$, where each β_i is associated with the appropriate constraints $\langle D_{\beta_i}, d_{\beta_i}, r_{\beta_i}, p_{\beta_i}\rangle$. A *feasible schedule* of a set of tasks $\mathcal{A}$ at time t_c represents a schedule S, such that its maximum lateness is less than or equal to zero. The *maximum lateness* of S, $L_{max}(S)$, is the $max_i\{f_{\beta_i} - d_{\beta_i}\}$, where f_{β_i} represents the time in which the execution of action $\beta_i \in S$ will be completed. The algorithm consists of two major steps. In the first step, it generates an initial schedule by using a *predictable Earliest Deadline First (EDF)* algorithm (Figure 16). In the second step, it tries to improve the schedule by using a simulated annealing technique (Figure 17). If the improving process also fails to find a feasible schedule, the RT scheduling subsystem sends the actions which caused the maximum lateness to be greater than zero back to the AI planning subsystem. Then the AI planning subsystem decides what to do with these actions. For example, it may decide to ask for help from other agents, or it may decide to change these actions' time requirements.

The algorithm is described in Figure 15. In the algorithm we assume that $\mathcal{A}=\{\beta_1, \ldots, \beta_n\}$ is the set of all the actions that should be scheduled which are known to the RT scheduling subsystem at a given time. This set is updated when new messages arrive from the AI planning subsystem. That is, the AI planning subsystem may send new actions, ask to delete

```
BUILD_INITIAL_SCHEDULE(t_sched)
11   S ← ∅;
12   while (A − S ≠ ∅)
13      A' ← all the ELIGIBLE actions at t_sched;
14      Select action β_i from A' which has earliest deadline;
15      Check if there is any task β_j ∈ A such that β_j
        is PREFERRED over β_i at t_sched;
16      if an action β_j is found then:
17         β_sched ← β_j;
18         else β_sched ← β_i;
19      if the time slot chosen for β_sched
        is occupied by a "constant action" β_const then:
20         β_sched ← β_const ;
21      schedule action β_sched into S;
22      t_sched ← f_{β_sched};
23   RETURN S;
```

Figure 16: The pseudo code of the EDF algorithm for building the initial schedule.

```
IMPROVE_SCHEDULE(S_best)
24   S_curr ← S_best;
25   B ← random value between 0 to 1;
26   for k ← 1 to K do
27      if message arrived from AI planning subsystem then RETURN;
28      if L_max(S_best) ≤ 0 then RETURN;
29      S' ← a randomly selected candidate successor of S_curr;
30      if L_max(S_best) < L_max(S') < L_max(S_curr) then:
31         S_curr ← S';
32      if L_max(S_best) > L_max(S') then:
33         S_best ← S'; S_curr ← S';
34      if L_max(S_curr) < L_max(S') then:
35         U ← random value between 0 to 1;
36         △E ← L_max(S_curr) − L_max(S_best);
37         if (U ≤ exp^{△E/B}) then:
38            S_curr ← S';
39         B_new ← random value less or equal than B
           but greater than 0;
40         B ← B_new ;
41   RETURN FAILURE;
```

Figure 17: The pseudo code of the simulated annealing algorithm, which is used in our system in order to improve the initial schedule.

some actions as a result of backtracking, ask to change some of the time requirements, and may also declare some actions to be "constant actions," i.e., actions that must be scheduled in a specific time slot (such "constant actions" may arise, for example, from a commitment to other agents to perform this action in a specific time).

In definition 4.2, we defined the relation "$\beta_j < \beta_i$" to be true if action β_j must be scheduled before action β_i, i.e., if a precedence relation between them exists. The *adjusted release time*, $r^{adj}_{\beta_i}$, of each action β_i is defined as follows:

$$r^{adj}_{\beta_i} = \begin{cases} r_{\beta_i}, \text{ if } \nexists \beta_j \text{such``} \beta_j < \beta_i'' \\ max(\{r_{\beta_i}\} \cup \{r^{adj}_{\beta_j} + D_{\beta_j} | \forall j, \text{``}\beta_j < \beta_i''\}), \text{otherwise;} \end{cases}$$

That is, the adjusted release time is the actual time of which the execution of an action β_i may begin as a result of the precedence constraints. In its first step, the algorithm calls the procedure BUILD_INITIAL_SCHEDULE($\mathcal{A}$) (Figure 16) to build an initial schedule S by using the predictable EDF algorithm. With this algorithm it first tries to build S from the actions with release times that have already been reached (i.e., greater than the current time t_c), but all of their preceding actions have been scheduled. These actions are called ELIGIBLE actions. Formally, we say that "action β_i is ELIGIBLE at a time point t" iff:
$(t \geq r^{adj}_{\beta_i}) \wedge \neg(f_{\beta_i} \leq t) \wedge (\nexists \beta_j \text{ such``} \beta_j < \beta_i'') \wedge (f_{\beta_j} > t)$.

In several cases, the arrival time of some action β_j may be earlier than its release time, since the RT scheduling subsystem receives its knowledge from the AI planning subsystem,. However, the algorithm will prefer to schedule action β_j before some other action β_i, although the release time of β_i has already been reached, since executing action β_i will cause β_j to miss its deadline. Formally, we say that "action β_j is PREFERRED over the ELIGIBLE action β_i at a time point t" iff:

$$\begin{array}{l} (r^{adj}_{\beta_j} > r^{adj}_{\beta_i}) \wedge (t + D_{\beta_i} > d_{\beta_j} - D_{\beta_j}) \wedge \\ (\nexists \beta_k \text{ such(``} \beta_k < \beta_j'')) \wedge (f_{\beta_k} > t) \end{array}$$

.

The procedure BUILD_INITIAL_SCHEDULE($\mathcal{A}$) also takes into consideration that some basic actions may be "constant actions." However, the procedure BUILD_INITIAL_SCHEDULE($\mathcal{A}$) does not guarantee that a feasible schedule will be found. Thus, if S, which was found, is unfeasible, the IMPROVE_SCHEDULE(S) procedure is initiated in the second step (line **9**) to try to improve the initial schedule. The best schedule found at any time is kept in S_{best}. In addition, another component of the RT scheduling subsystem is a dispatcher that executes the actions of S_{best} according to their appropriate start time. When an action is executed by the dispatcher, it is deleted from the schedule S_{best} and from $\mathcal{A}$.

The IMPROVE_SCHEDULE procedure runs, at most, K iterations[9] and tries to improve the initial schedule using a simulated annealing search algorithm [55]. It builds the "candidate successor" schedule S' of S_{curr}, by selecting 2 random actions (which are not constant) from S_{curr} and by interchanging them. Then the maximum lateness, $L_{max}(S')$, of the "candidate schedule" is calculated, and if it is lower than the maximum lateness, $L_{max}(S_{best})$, of the best possible schedule found so far, it will become the new best possible schedule. In some cases, we will choose to adopt a "candidate schedule" whose lateness is larger than our best known schedule. This is because we wish to avoid local minima. The chances of reverting to a bad schedule are determined by a probability function that takes into account the number of iterations, as well as our best known lateness. Therefore, the chances of reverting to a very bad candidate schedule are slim, and will decrease even more as the iterations of the function progress. This is achieved through the parameter B, which receives an initial value

[9]In our simulations, the value of K was 400 when a backtracking was allowed in the system, but when the backtracking was not allowed, K was 10000.

and decreases as the iterations progress, thus limiting the possibility of reverting to a bad schedule.

The IMPROVE_SCHEDULE procedure stops before it performs K iterations if it manages to find a schedule in which the lateness is less than or equal to zero (line **28**), or if a message is received from the AI planning subsystem (line **27**). In the first case, S_{best} will be used by the dispatcher (which is not described in the figure.) In the case of a message from the AI planning subsystem, the RT scheduling subsystem will update the set $\mathcal{A}$ (line **5**), and the procedure BUILD_INITIAL_SCHEDULE(T) will construct a new initial schedule for the new set of actions, taking into account constant actions. The IMPROVE_SCHEDULE notifies of failure after K iterations because no feasible schedule was found. In this case, the algorithm identifies the actions that miss their deadline and reports them to the AI planning subsystem (line **10**). The AI planning subsystem either backtracks and searches for a different plan or requests help from other agents.

It is easy to show that the order of the actions in S_{best} satisfies the release time and the precedence constraints, since the algorithm schedules the actions according to their partial order. That is, it first schedules the first action in the partial order, and so on. It also schedules the actions in such a way that their start time in the schedule is greater than their release time. As a result, if $L_{max}(S_{best}) \leq 0$, then BUILD_FEASIBLE_SCHEDULE builds a schedule which satisfies all of the temporal requirements of the actions in $\mathcal{A}$.

7 Experimental Results

In this section we present the results of the experiments conducted using our system in a simulation environment. In our simulations, we ran the algorithm on several different recipe libraries which were created randomly. Each recipe library stratifies the following: (1) the agent is able to build at least one "complete recipe tree" for action α, which constitutes performing α under its associated time constraints θ_α, and (2) there is at least one feasible schedule which may consist of the all basic actions in the "complete recipe tree" of α and their associated time constraints.

We assert that the agent succeeded in performing the highest level action α if: (1) the AI planning subsystem finished the planning for α and built the complete recipe tree for α which satisfies the appropriate time constraints θ_α; and if (2) the RT scheduling subsystem found a feasible schedule, which consists of all the basic level actions in the complete recipe tree of α. A failure is defined as either a failure in the planning by the AI planning subsystem, or as a failure in finding a feasible schedule by the RT scheduling subsystem. In the following experiments, we make the simplifying assumption that all of the time constraints of the agent's action are associated either with the action-type or with its appropriate recipe.

In our simulation environments we built the recipe library according to the following parameter settings: (1) the average number of the *precedence constraints* between subactions in each recipe in the recipe library; (2) the average number of the *metric constraints* of all the actions in the recipe tree; (3) the *depth* of the recipe tree; (4) the number of the total *basic actions* in the recipe tree, where the duration of each basic action may be between 1 to 10 minutes; and (5) the *idle time*, which indicates the average time (in seconds) between two basic actions that may be "empty" in the scheduler.

In order to study the implications of the above parameters, we ran 4 sets of experiments. The goal of the first set of experiments was to identify environments in which our system always succeeds. In the second set of experiments we study how different parameters influence the performance of our algorithm. After verifying in the second set of experiments that the number of the basic level actions in the full plan and the average idle time are the most influential parameters, we ran a third set of experiments. In this set of experiments we studied the average idle time that is necessary for a given number of basic actions to achieve a high percentage of success. In the first, second, and third sets of our experiments, we checked the algorithm in a simple environment of a single agent without the possibility of backtracking. In the fourth set, we enabled the agent to backtrack, while the goal of the experiments, was to test the performance of the system when the agent is able to backtrack. The results of all the experiments are described below.

7.1 Experiments with Single Agent without Backtracking

In the first set of experiments, we ran 100 examples. The examples are created as described above. Our experiments proved that if the execution of all the basic actions begins 60 seconds or more after the planning process has started, and if the total number of the *basic actions* is less than 70, then the system almost always succeeded (close to 100%), given that the idle time is at least 120 seconds. In addition, if the *ideal time* is very high (more than 500 seconds), the system always succeeds, regardless of the values of the other parameters, given that the system does not need to start the planning and the execution simultaneously. Similarly, we attained a 100 percent rate of success when the average number of *metric constraints* was high (an average of at least 1.4 constraints per action), regardless of the values of the other parameters, given that the system does not need to start the planning and the execution simultaneously.

The major goal of the second set of the experiments was to check how the value of each parameter influences the success rate of our system. Thus, in this set, we did not consider environments that were identified in the first set of experiments as environments in which the system almost always succeed. We ran 246 experiments. In each run, the above parameters were drawn from the following given range: (1) the average number of the *precedence constraints* was between 0 and 12; (2) the average number of the *metric constraints* was between 0 and 18; (3) the *depth* of the recipe tree was between 0 and 9; (4) the number of the total *basic actions* in the recipe tree was between 60 and 120; and (5) the *idle time* was between 50 and 500 seconds. In addition, in all the runs there was no delay between the start time of the planning and the beginning of the execution time. That is, the release time of some of the basic level actions was concurrent with the time of the beginning of the planning.

The graph in Figure 18 presents the success rate we attained for each given range of basic tasks in the "complete recipe tree." As shown in the graph, the success of the system depends on the number of the basic actions in the "complete recipe tree." When the number of the basic actions was low (70-80 actions), the success rate was about 76 percent. For an intermediate number of basic actions of 80-90, the success rate was about 58 percent, and for 90-100 basic actions, the success rate decreased to 53 percent. For a high number

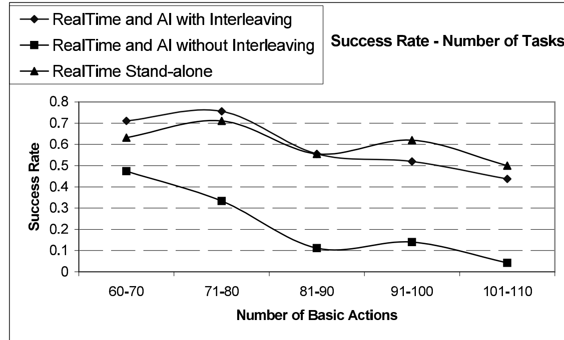

Figure 18: The graph "RealTime and AI with interleaving" describes the success rate of our system, which interleaves planning and execution. The "RealTime and AI without interleaving" graph describes the performance of the system when the AI planning subsystem first finishes the entire planning process for the action. Then, the scheduling and execution process is run on the basic actions. The "RealTime stand-alone" graph describes the performance of the RT scheduling stand alone system when no planning is necessary.

of basic actions, the success rate was 43 percent. The reason why we attained a lower rate (62 percent) of success in the first range of basic actions (60-70) is that the planner could send the first tasks to the RT scheduling subsystem very quickly (in a matter of seconds). As a result, the RT scheduling subsystem started the execution of these actions. However, since our system performs non-preemptive basic-actions, we encountered situations in which we started to execute basic actions that could have potentially been scheduled much later. However, since they were the only available tasks at the moment, and since their release time allowed us to begin their schedule, it caused other tasks, that were received several seconds later, to be scheduled later. In turn, this caused a failure in those runs. But, if we provide the schedule with a waiting time at the beginning of the schedule process (i.e., all the basic actions which arrive at the RT scheduling subsystem have to begin 60 seconds (or more) after the schedule process has been started), then the success rate of (60-70) basic actions is 100 percent. However, if the AI planning subsystem would have always advised the RT scheduling subsystem to delay the execution of the first basic level actions in 60 seconds, the performance of the system when there are more than 70 basic actions will decrease significantly.

To compare our method to other approaches that do not interleave planning and execution, the agent was not allowed to start the execution before a full plan was available. Thus, in this experiment, we first ran the AI planning subsystem. Then, immediately after the AI planning subsystem finished planning all the action, the scheduling algorithm (of the RT scheduling subsystem) was operated on the basic level actions and their associated time constraints (as in static environments). The outcome is given in the graph of Figure 18. As indicated in the graph, the success rate of the case when the system has started the execution only after a full plan had been identified is very low (less than 50 percent). The reason for the low success rate is result from the fact that a high number of basic tasks (about 100 actions)

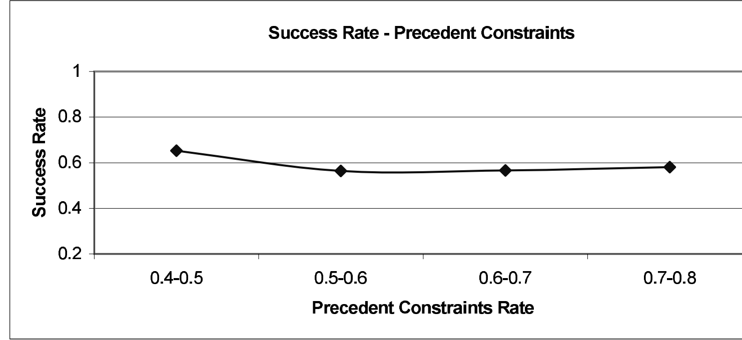

Figure 19: Success rate - average number of precedent constraints.

cause the planning time to be high. Since the AI planning subsystem first completed the planning and only then sent the basic actions to the RT scheduling subsystem, some basic actions missed their deadline and the RT scheduling subsystem was unable to schedule them under their temporal constraints. Thus, it is clear that interleaving planning and execution is crucial in such environments.

In order to evaluate the performance of our RT scheduling subsystem, which is based on a heuristic algorithm, we tested the success rate of the RT scheduling stand-alone system on the same data, assuming that no planning is necessary. In this experiment, we ran the scheduling algorithm (of the RT scheduling subsystem) without the planning process. That is, all the basic actions were sent to the RT scheduling subsystem as an input at the beginning of its scheduling process. To our surprise, as indicated from the graph, up until a level of 80 actions, the combined system that consists of both planning and execution is better than the RT scheduling stand-alone system. We believe that this is because the AI planning subsystem sends the basic actions to the RT scheduling subsystem according to their execution order. This helps the RT scheduling subsystem identify a feasible schedule. We conclude that when the number of basic actions is low, the planning time is small and does not delay the overall execution. This conclusion extends the results of Knblock's work [31] which built a system integrating planning and access to external information source (called "sensing"). His experiments show that in his domain of information gathering, the planning time is only a small fraction of the overall execution time. But his work considers only scenarios with 10 actions maximum, while our results are correct for to up to 110 actions. However, when the number of the basic actions increases to more than 110, the RT scheduling stand-alone system outperforms our system. In such environments, the high planning time that leads to a delay in sending the basic actions to the RT scheduling is more influential on the success rate than is the implicit information provided to the RT scheduling subsystem by receiving the actions in correct order.

Figure 19 shows the success rate we had in a given range of precedence constraints. The "Precedent Constraints Rate" axis represents the average number of precedence constraints per action in the complete recipe tree. We can see that the trend of the line is a straight line, with a success rate that reaches almost 60 percent. We assume that the reason why the

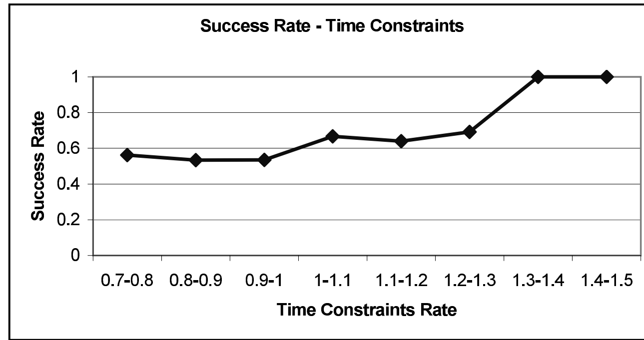

Figure 20: Success rate - average number of time constraints.

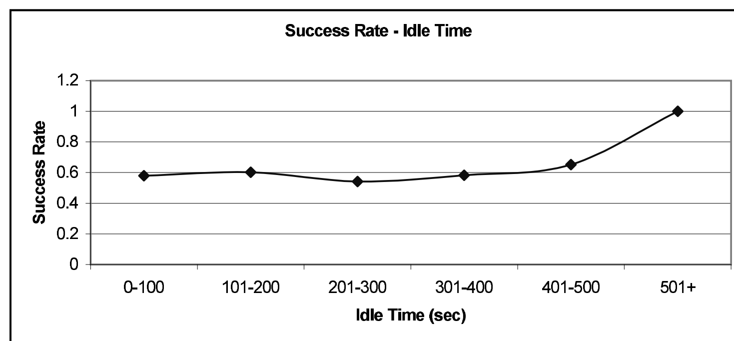

Figure 21: Success rate - idle-time.

success rate does not really change, even though the rate of precedence increases, is because the precedence constraints cause the scheduling problem to be more difficult. On the other hand, the precedence constraints provides more knowledge about the subactions slots. As a result, the precedence constraints leads the scheduler to the correct solution (which always exists in our examples).

The graph in Figure 20 presents the success rate as a function of the metric constraints. When the average number of the metric time constraints per either complex or basic level action in the "complete recipe tree" increases, the success rate of our system also increases. The reason for this surprising result is that the metric time constraints provide the scheduler with more precise knowledge about the scheduling slot of the basic actions.

The last parameter which was tested in this set of experiments is the idle-time. As presented in the graph of Figure 21, when the idle-time was high (greater than 500 seconds) the success rate was 100 percent. However, when the idle time was less than 500 seconds the success rate was about 60 percent. It is clear that high idle time allows for a better method of scheduling and more manipulation of the scheduled tasks.

The reason why we did not attain a 100 percent success rate in most of the experiments of this set stems from the fact that when we tested the performance of one parameter we

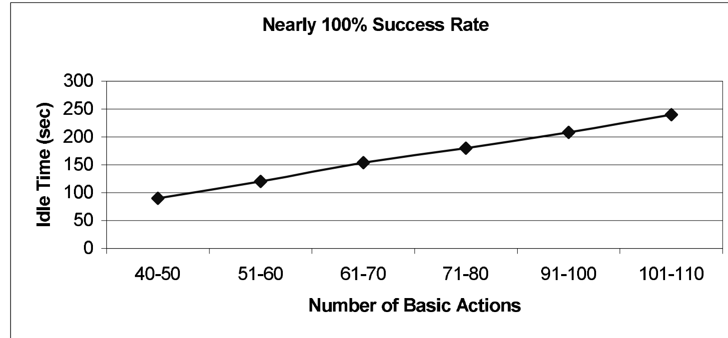

Figure 22: The average idle time that is necessary for a given number of basic actions for a high percentage of success rate.

allowed the other parameters to have different values. However, the above results prove that the major parameters which influence the success of the system are the *number of basic actions* and the *idle time*. Thus, we ran a third set of experiments to check which configuration set of these parameters attained about a 100 percent success rate. For this purpose, we ran 120 examples (20 examples for each configuration). Figure 22 presents the minimal idle time that is needed to attain about 100 percent success rate as a function of the number of basic actions in the recipe tree. The values of the other parameters in this set were drawn from the same range as the second set of experiments described above. As illustrated in the results' graph in Figure 22, when the number of basic level actions is small, the idle time may be low. However, when the number of basic actions is large, the planning process becomes slower. As a result, the idle time should be higher in order to guarantee about a 100 success percent rate.

7.2 Single Agent Backtracking

In this set of experiments, we tested the effects of backtracking by allowing more than one recipe for each complex level action in the recipe tree of α. We ran 43 different random examples of recipe libraries for action α. We changed each example of a recipe library 9 times, as follows. At the beginning, the recipe library included only one recipe for each complex level action (as in the case that backtracking was not allowed). Then, one complex action had two recipes, two actions had two recipes, and so on (we increased the number of recipes by one 9 times). The examples were drawn from environments with a high success rate the value of the *number of basic actions* was between 50 and 70, and the value of the *ideal time* was between 80 and 150 (the configurations set of these values were selected according to the graph in Figure 22). The values of the other parameters were randomly selected from the same range as the two former sets of experiments. The results of this experiment are presented in Figure 23. It is clear that the backtracking causes the planning time to be higher, thus the performance of the system decreases. On the other hand, when the number of the additional recipes is high, in several cases, the agent has additional options

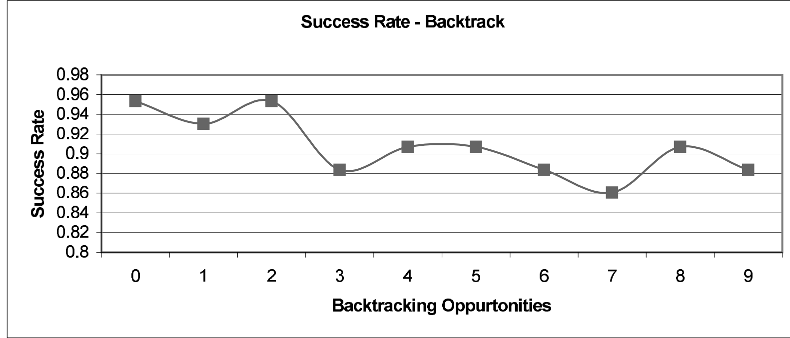

Figure 23: Success Rate - in backtracking environment.

in the performance its plan, which may cause the system's performance to improve. Since our system interleaves planning and execution, in the backtracking, the agent may remove a recipe which consists of subactions that already have been executed. In our experiments, we analyzed the number of the basic actions that had been executed and then were removed from the agent plan. We found that in most cases (94 percent), the agent removed actions that had not been performed yet. However, we did not consider the effects of the executed actions which were removed.

8 Expanding the Algorithm for the Multi-Agent Activity

Although several collaborative multi-agent systems have been developed, they pay little attention to time coordination problems. There are mainly two approaches. The first approach suggests a synchronization method; i.e., the team members broadcast messages that ensure that the execution of an action will not be attempted until all the team members are ready [59, 65]. In the second approach [27], a team leader is responsible for the timing of the individual actions. Both approaches restrict the activity of the individual agents, and the second approach is not appropriate for uncertain dynamic environments.

The temporal reasoning mechanism for the multi-agent environment is very similar to that of the individual case. Each individual agent G maintains a temporal constraints graph and develops its graph during its planning. However, in this case, the leaves of the full recipe tree may be complex actions to be performed by other agents. In particular, some vertices in the incremently constructed graph may represent joint actions. As a result, each agent maintains a different graph which consists of its constraints. Thus, Gr_α, where α may be either a joint action or an individual one of some agent G, could include five types of vertices: (1) vertices which denote complex actions that G needs to perform individually; (2) vertices which denote basic actions that G needs to execute; (3) vertices which denote complex actions, where G belongs to the groups which have to perform the actions; (4) vertices that denote multi (or single) agent actions and where G does not belong to the

groups (or is not the agent) that have (has) to perform these actions, but where the value of these vertices are already known; and (5) vertices that denote multi (or single) actions, where G does not belong to the groups (or is not the agent) which have (has) to perform the actions and where the value of these vertices is unknown. When a vertex status is changed from unexplored to explored, the algorithm also determines to which of the cases described above it belongs.

The concept IndV is also different from the individual case, because each vertex may depend on either other actions or other agents. In the multi-agent version of the algorithm, the first and the second types of vertices will be developed as in the individual case. In the development of the third type, the agents have to exchange their time constraints on the joint action, and all agents need to agree to the set of constraints. Then the agents use the planning process **Select_Rec_GR** to jointly select the recipe for the joint action and to assign subactions to individuals and subgroups. Based on these decisions, each agent constructs the $Gr^{p}_{R_\alpha}$ graph of the joint action, incorporates it in Gr_α, and continues its planning process. We refer to the fourth type of vertex as fixed (with the assumption that the agent may be asked to change the value). These values have been sent to the agent before it reaches this state of its planning. The last type influences which vertices are independent; i.e., if some of the ancestors of a certain vertex is of this type, then the agent can not continue the development of this vertex until it receives the value of the vertex from the appropriate agents.

To confirm the correctness of the multi-agent version of the algorithm, we have proved that no deadlocks occur, since at each stage of the planning process at least one of the agents in the group has a non-empty IU [24].

9 Conclusions

In this paper we have presented a mechanism for time planning in uncertain and dynamic multi-agent systems, where the agents interleave planning and acting. This mechanism allows each agent to develop its timetable individually and to take into consideration all types of time constraints on its activities and on these of its collaborators. Thus, in contrast to other works on multi-agent systems, which suggest that either the team members maintain full synchronization by broadcasting messages or that a team leader determines the times of actions, our mechanism enables the individual agent to determine its time individually, but also to be easily integrated in the activities of other agents. We have proved that under certain conditions the mechanism is sound and complete.

We have also presented the results of several experiments on the system. The results proved that interleaving planning and acting in our environment is crucial. In our experiments, we tested the influence of several parameters on the system's performance. We showed the configuration set of the parameters for the system provides an almost 100% success rate. We found out that as the number of the basic actions increases, the performance of the system decreases. However, this performance may be improved by improving the efficiency of the algorithm in the AI planning subsystem. We have left the task of improving the algorithm for future work.

The development of this mechanism has uncovered several interesting problems in designing agents for collaborative activity. First, the planner does not consider the preconditions of the actions while developing its plan. Second, the planner does not reason about resource and resource conflicts in the context of time coordination. We also did not consider the side effects which arise from executing the actions that were canceled as a result of the agent backtracking. We have left each of these research problems for future work. The next major steps that we envision includes developing strategies and protocols for elaborating partial plans, including mechanisms for combining time information possessed by different agents, as well as strategies for negotiating in the event of resource conflicts, and reaching a consensus on how to allocate portions of the activity among different participants.

References

[1] James Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–144, 1984.

[2] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26:832–843, 1983.

[3] R. C. Arkin. Integrating behavioal, perceptual, and world knowledge in reactive navigation. *Robotics and Autonomous Syst.*, 6:105–222, 1990.

[4] R. Arthur and J. Stillman. Tachyon: A model and environment for temporal reasoning. M.S. Thesis, Rensselaer Polytechnic Institute, Troy, NY, 1992.

[5] Peter Van Beek. Reasoning about qualitative temporal information. *Artificial Intelligence*, 58:297–326, 1992.

[6] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems.* Kluwer Academic Publishers, 1997.

[7] Alison Cawsey. *Artificial Intelligence.* Prentice Hall Europe, London, 1998.

[8] S. Chien, G. Rabideau, J. Willis, and T. Mann. Automating planning and scheduling of shuttle payload operations. *Artificial Intelligence Journal*, 114:239–255, 1999.

[9] Wesley W. Chu and Kin K. Leung. Task response time for real-time distributed systems with resource contentions. *IEEE Transactions on software engineering*, 17(10):1076–1092, 1991.

[10] Thomas H. Cormen, Charles E. Leiserson, and Ronland L. Rivest. *Introduction to Algorithms.* MIT Press, London, England, 1990.

[11] Ken Currie and Austin Tate. O-plan: the open planning architecture. *Artificial Intelligence*, 52:49–86, 1991.

[12] C. Anderson D. Weld and D. Smith. Extending graphplan to handle uncertainty and sensing actions. In *Proceedings of the Annual Conference of the American Association for Artificial Intelligence (AAAI-98)*, pages 897–904, 1998.

[13] T. L. Dean and D. V. McDermott. Temporal data base management. *Artificial Intelligence*, 32:1–55, 1987.

[14] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[15] J. Dorn, M. Girsch, G. Skele, and W. Slany. Comparison of iterative improvement techniques for schedule optimization. *European Journal on Operations Research*, pages 349–361, 1996.

[16] A. El-Kholy and B. Richards. Temporal and resource reasoning in planning: the parcPlan approach. In *12th European Conference on Artifical Intelligence (ECAI '96)*, pages 614–618, 1996.

[17] R. E. Fikes, P.E. Hart, and N. J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4):251–288, 1972.

[18] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, San Francisco, California, 1979.

[19] Keith Golden, Oren Etzioni, and Daniel Weld. Planning with execution and incomplete information. Technical report, University of Washington, 1996.

[20] Martin Charles Golumbic. Reasoning about time. *Mathematical Aspects of Artificial Intelligance*, 55:19–53, 1998. Proceedings of Symposia in Applied Mathematics.

[21] M. Graham and M. Florian. On scheduling with ready times and due dates to minimize maximum lateness. *Operations Research*, 23(3), 1975.

[22] Barbara Grosz and Sarit Kraus. Collaborative plans for group activities. In Ruzena Bajcsy, editor, *Proceedings of the 1993 International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 367–373, San Mateo, CA, 1993. Morgan Kaufmann Publishers, Inc.

[23] Barbara J. Grosz and Sarit Kraus. Collaborative plans for complex group action. *Artificial Intelligence*, 86(2):269–357, 1996.

[24] Meirav Hadad and Sarit Kraus. Intelligent information agents. In Matthias Klusch, editor, *Intelligent Information Agents*, chapter SharedPlans in Electronic Commerce, pages 204–231. Springer Publishers, Berlin, 1999.

[25] Babak Hamidzadeh and Shashi Shekhar. Specification and analysis of real-time problem solvers. *IEEE Trans. on Computer*, 19(8):788–803, August 1993.

[26] Kristian J. Hammond. Chef: A model case-based planning. In *Proceedings of the Annual Conference of the American Association for Artificial Intelligence (AAAI-86)*, pages 267–271, 1986.

[27] Nick R. Jennings. Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. *Artificial Intelligence*, 75(2):1–46, 1995.

[28] J. Hendler K. Erol, D. Nau. HTN planning: Complexity and expressivity. In *Proceedings of the Annual Conference of the American Association for Artificial Intelligence (AAAI-94)*, pages 1123–1128, 1994.

[29] Nikos I. Karacapilidis. Planning under uncertainty: A qualitative approach. *Lecture Notes in Computer Science*, 990:285–296, 1995.

[30] D. Kinny, M. Ljungberg, A. S. Rao, E. Sonenberg, G. Tidhar, and E. Werner. Planned team activity. In C. Castelfranchi and E. Werner, editors, *Artificial Social Systems, Lecture Notes in Artificial Intelligence (LNAI-830)*, Amsterdam, The Netherlands, 1994. Springer Verlag.

[31] Craig A. Knoblock. Building a planner for information gathering: A report from the trenches. In *Third International Conference on Artificial Intelligence Planning Systems*, 1996.

[32] Carsten Hojmose Kristensen and Niels Drejer. Evaluating distributed timing constraints implementing run-time mechnisms. In *2ND IEEE workshop on Real-Time Applications*, pages 157–162, Washington,DC., 1994.

[33] John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[34] J. Lever and B. Richards. parcPlan: a planning architecture with parallel actions, resources and constraints. *Lecture Notes in Computer Science*, 896:213–222, 1994.

[35] H. Levesque, P. Cohen, and J. Nunes. On acting together. In *Proceedings of the Annual Conference of the American Association for Artificial Intelligence (AAAI-90)*, pages 94–99, 1990.

[36] Karen Lochbaum. *Using Collaborative Plans to Model the Intentional Structure of Discourse.* PhD thesis, Harvard University, 1994. Available as Tech Report TR-25-94.

[37] Karen Lochbaum, Barbara Grosz, and Candace Sidner. Models of plans to support communication: An initial report. In *Proceedings of AAAI-90*, pages 485–490, Boston, MA, 1990.

[38] J. Malik and T.O. Biford. Reasonning in time and space. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-83)*, pages 343–345, 1983.

[39] Graham Mcmahon and Michael Florian. On scheduling with ready times and due dates to minimize maximum lateness. *Operations Research*, 23(3):475–482, 1975.

[40] Pinedo Michael. *Scheduling.* Prentice Hall, 1995.

[41] D. P. Miller and E. Gat. Exploiting known topologies to navigate with low-computation sensing. In *SPIE Sensor Fusion Conference*, November 1990.

[42] U. Montanari. Networks of constraints: fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.

[43] L. Morgenstern. Knowledge preconditions for actions and plans. In *Proceedings of IJACAI-87*, pages 867–874, Los Altos, CA, 1987. Morgan Kaufman Publishers Inc.

[44] D. J. Musliner, E. H. Dufree, and K. G. Shin. World modeling for dynamic construction of real-time control plans. *AI*, 74(1):83–127, 1995.

[45] David J. Musliner, Edmund H. Dufree, and Kang G. Shin. Circa: A cooperative intelligent real-time control architecture. *IEEE trans. on computer*, 23(6):1561–1574, November 1993.

[46] David J. Musliner, James A. Hendler, Ashok K. Agrawala, Edmund H. Dufree, Jay K. Strosnider, and C. J. Paul. The challenges of real-time AI. *IEEE trans. on computer*, 28(1):58–66, January 1995.

[47] Alexander Nareyek. A planning model for agents in dynamic and uncertain real-time environments. In *AIPS Workshop on Integrating Planning*, pages 7–14, 1998.

[48] Alexander Nareyek. Open world planning as scsp. Technical Report WS-00-02, AAAI Press, Menlo Park, California, 2000.

[49] Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. Shop and m-shop planning with ordered task decomposition. Technical report, University of Maryland, 2000.

[50] Nils J. Nilsson. *Principles of Artificial Intelligence*, chapter 7 and 8. Tioga Publishing Company, 1980.

[51] J.S. Penberthy and D. Weld. Temporal planning with continuous change. In *Proceedings of the Annual Conference of the American Association for Artificial Intelligence (AAAI-94)*, pages 1010–1015, 1994.

[52] Martha E. Pollack. Plans as complex mental attitudes. In P.N. Cohen, J.L. Morgan, and M.E. Pollack, editors, *Intentions in Communication.* Bradford Books, MIT Press, 1990.

[53] Krithivasan Ramamritham and John A. Stankovicand. Dynamic task scheduling in hard real-time distributed systems. *IEEE Software*, 1(3):65–75, 1984.

[54] Elaine Rich and Kevin Knight. *Artificial Intelligence.* McGraw-Hill, Inc., second edition, 1991.

[55] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Englewood Cliffs, NJ, 1995.

[56] Terry Shepard and J. A. Martin Gagne. A pre-run-time scheduling algorithm for hard real-time systems. *IEEE Transactions on software engineering*, 17(7):669–677, 1991.

[57] R. Simmons. An architecture for coordinating planning, sensing and action. In *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling, Contr.*, pages 292–297, November 1990.

[58] Hardeep Singh and Texas Instrument. Scheduling techniques for real-time applications consisting of periodic task sets. In *2ND IEEE Workshop on real-time applications*, pages 12–15, Washington,DC, 1994.

[59] E. Sonenberg, G. Tidhar, E. Werner, D. Kinny, M. Ljungberg, and A. Rao. Planned team activity. Technical Report 26, Australian Artificial Intelligence Institute, Australia, 1992.

[60] Biplav Srivastava. RealPlan: Decoupling causal and resource reasoning in planning. In *Proceedings of the Annual Conference of the American Association for Artificial Intelligence (AAAI-00)*, pages 812–818, 2000.

[61] J. A. Stankovic and K. Ramamritham. The spring kernal: a new paradigm for real-time systems. *IEEE Software*, pages 62–72, May 1991.

[62] John A. Stankovic, Krithivasan Ramamritham, and Shengchang Cheng. Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems. *IEEE Transactions on computers*, 34(12):1130–1143, 1985.

[63] John A. Stankovic, Marco Spuri, Macro Di Natale, and Giorgio Buttazzo. Implications of classical scheduling results for real-time syatems. *IEEE computer*, 28(6):16–25, 1995.

[64] Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, 1990.

[65] M. Tambe. Toward flexible teamwork. *Journal of AI Research*, 7:83–124, 1997.

[66] Steven A. Vere. Planning in time: Windows and durations for activities and goals. In James Allen et al., editor, *Readings in Planning*, pages 297–318. Springer Publishers, Berlin, 1990.

[67] T. Vidal and M. Ghallab. Dealing with uncertain durations in temporal constraint networks dedicated to planning. In *12th European Conference on Artifical Intelligence (ECAI '96)*, pages 54–48, 1996.

[68] M. Vilain and H. A. Kautz. Constraint propagation algorithms for temporal reasoning. In *Proceedings of the Annual Conference of the American Association for Artificial Intelligence (AAAI-86)*, pages 132–144, 1986.

[69] D. Wilkins. Using pattern and plans in chess. *Artificial Intelligence*, 14(2):165–203, 1983.

[70] D. E. Wilkins, K. L. Myers, J. D. Lowrance, and L. P. Wesley. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical AI*, 7(1):197–227, 1995.

[71] Jia Xu. Multiprocessor scheduling of process with release times, deadlines, precedence, and exclusion relations. *IEEE Transactions on software engineering*, 19(2):139–154, 1993.

[72] Jia Xu and David Lorge Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. *IEEE Transactions on software engineering*, 16(3):360–369, 1990.

[73] Jia Xu and David Lorge Parnas. On satisfying timing constraints in hard-real-time systems. *IEEE Transactions on software engineering*, 19(1):70–84, 1993.

A Lemmas and Propositions for the Correctness of the Algorithm

In order to prove the correctness of the algorithm, we will first define the following new concepts. All of the concepts refer to the algorithm and the definitions in section 4.2.

Definition A.1 (CloserFixVertex(v), CloserFixSet(v)) *Let $PathSet(v)$ be the set $\{p_1, \ldots, p_m\}$ which consists of all the paths from $s_{\alpha_{plan}}$ to the vertex $v \in V_\alpha$. For each $p_i \in PathSet(v)$ $(1 \leq i \leq m)$, the fixed vertices in p_i which is closer to v, is called the $CloserFixVertex(v)$ in p_i. That is, if p_i is the sequence $(u_0, u_1, u_2, \ldots, u_n)$ such that $s_{\alpha_{plan}} = u_0$ and $v = u_n$, then, u_j $0 \leq j \leq n$ is the $CloserFixVertex(v)$ in p_i if for all other fixed vertex u_s in p_i $s < j$. The $CloserFixSet(v)$ is a set of all the vertices $\{v_1, v_2, \ldots, v_l\}$ such that each v_j $1 \leq j \leq l \leq m$ is a $CloserFixVertex(v)$ in some p_i.*

In each stage of the algorithm, the agent selects a vertex v which is an IndV in order to identify the value of this vertex. In the following claims we will show that, according to the algorithm, this vertex indeed can be identified (because all the vertices in the path from the vertices in the $CloserFixSet(v)$ to v have been identified). Then, we will show (in lemma 1) that the algorithm always terminates, and finally we will prove that the algorithm is correct and complete.

Proposition 1 *Suppose that* IDENTIFY_ID_TIMES_PARAMS *builds a temporal constraints graph of α, $Gr_\alpha = (V_\alpha, E_\alpha)$, from a given initial graph $InitGr_\alpha$. Let u be some vertex in V_α. Then, during the execution of* IDENTIFY_ID_TIMES_PARAMS, *if each vertex in the path from all the vertices in the $CloserFixSet(u)$ to u are explored (i.e., either EB or EC), then u is IndV.*

Proof: The proof is by induction on the length of the longest path from $CloserFixSet(u)$ to u. Initially, when the vertex u is a fixed time point, the proposition certainly holds. Let u be some vertex such that the path from all the vertices in the $CloserFixSet(u)$ to u are explored. If u is not a fixed vertex, then for each $(v, u) \in E_\alpha$ the $CloserFixSet(u)$ is longer than the $CloserFixSet(v)$. Thus the path from all the vertices in the $CloserFixSet(v)$ to v is explored. Thus, according to the inductive hypothesis, each such v is IndV; that is, $v \in IE$. Now, it remains to show that the weight of each edge $(v, u) \in E_\alpha$ is known. Since, according to the algorithm, this edge cannot be a complex edge,[10] then the weight of this edge is known. As a result, the weight of the edge from the $CloserFixSet(v)$ to v is known and the value of v may be defined (i.e., v is IndV). □

In the following proposition we prove that all the values of the parameters associated with the basic actions which are sent to the RT scheduling subsystem will not be changed later by the planning process (with the exception of the case of backtracking). Thus, these actions can already be scheduled and executed before the agent finishes its planning. This fact enables the agent to interleave planning and acting.

[10]That is, in the building of the graph, it first adds the new vertices which are appropriated for the selected recipe of the complex edge (v, u) between u and v. Then, it changes the statues of v and u to explored vertices. As a result, there is path of unexplored vertices between u and v.

Proposition 2 *Suppose that* IDENTIFY_ID_TIMES_PARAMS *runs and builds a temporal constraints graph of α, $Gr_\alpha = (V_\alpha, E_\alpha)$. Let $v \in V$ be an IndV which represents a start time point of some basic level action β. Then, the final values of $< C_\beta, d_\beta, r_\beta, p_\beta >$ can be identified, i.e.; these values will not be changed (unless the agent will backtrack and choose a different recipe).*

Proof: Because β is a basic level action it is obvious that the computation time, C_β, is final.[11] Now, we have to show that if v represents a start time point of β, and v is an IndV, then r_β can be identified. Since v is IndV all the paths between the $CloserFixSet(v)$ to v are final, the weights of all the edges in these paths are final, thus the final value of r_β can be identified. Similarly, the final value of d_β can be identified. Also, all the basic edges in the paths between the $CloserFixSet(v)$ to v are final. Thus all the basics actions which are precedent to v are final.□

In the following lemma we prove that the algorithm always terminates.

Lemma 1 *Let $Gr_\alpha = (V_\alpha, E_\alpha)$ be the temporal constraints graph of α, and suppose that* IDENTIFY_ID_TIMES_PARAMS *runs and builds Gr_α from a given initial graph $InitGr_\alpha$. Then, during its execution:*

1. *If Gr_α consists of some unexplored vertices, then at least one of these vertices is an IndV start time point.*

2. *If all the vertices in the graph are explored, then the agent finished identifying all of the time parameters in its individual plan for α.*

Proof:

1. Because Gr_α is a "directed acyclic graph (dag)," we can perform a topological sort of the graph. Let v_i be the first unexplored vertices in the order of the topological sort. Then, it is clear that all the vertices in the paths from $s_{\alpha_{plan}}$ to v_i are explored. Thus, according to proposition 1, v_i is an IndV. According to the algorithm, the start time point and the finish time point of each action in the graph become unexplored in the same step. The start time of the action always precedes the finish point. Thus, v_i is a start time point (since the start time point is also prior to the finish time point in the topological order).

2. Suppose by contradiction that the agent did not finish identifying all of the temporal parameters in its individual plan for α, but all the vertices in the graph were explored. Thus, the graph consists of at most one action β for which the agent did not identify its time parameters. But, according to the algorithm, for each basic level action β, the vertices which represent action β become explored when the temporal parameters of β are identified. For each complex level action β, when the vertices which represent this action become explored, new unexplored vertices are added to the graph (i.e., the vertices which represent the subactions of β). Thus, we obtain a contradiction to the assumption that all the vertices in the graph are explored.□

[11] We use the term "final" to refer to the values of the parameters that will not be changed during the planning process (unless the agent backtracks).

Corollary 1 *The algorithm terminates when all the leaves in the implicit recipe tree of Gr_α are basic level actions.*

Proof: Otherwise, if the leaf of the recipe tree was a complex level action, then, according to the algorithm, the graph includes some unexplored vertex and the algorithm has not terminated.

In the following lemma we prove that in each stage of the algorithm the leaves in the implicit recipe tree constitute performing α under the associated time constraints α.

Lemma 2 *Suppose that the agent plans for some action α. Then, during the development of the graph Gr_α, performing all the actions (possibly complex) in the leaves of the implicit recipe tree, $Tree_\alpha$, constitutes performing α under constraints θ_α (possibly in parallel).*

Proof: The proof is by induction of the number of the vertices in the implicit recipe tree. Initially, when the recipe-tree consists of one vertex (i.e., only action α), the proposition certainly holds. Suppose that the recipe tree consists of n vertices. As an inductive hypothesis we assume that all the actions in the leaves of the recipe tree with m vertices, where $m < n$ constitute performing α under constraints θ_α. Let $Tree_\alpha$ be some incomplete recipe tree of α with m vertices. According to the algorithm the agent expands the recipe tree by selecting some leaf which represents a complex level action β. Suppose that the agent selected the recipe R_β in order to expand the recipe tree for α which consists of b subactions. According to the inductive hypothesis, all of the actions in the leaves of the recipe tree with the m vertices constitute performing α under constraints θ_α. However, according to the algorithm, when the agent selects the recipe for β and adds it to the constraints graph, it also checks that all the constraints of β constitute performing α under constraints θ_α. If these constraints constitute performing α, it continues with its plan for α and the subactions of β become the leaves of the recipe tree of α. Thus, all the actions in the leaves of the tree which consists of $n = m + b$ vertices constitute performing α under constraints θ_α. □

B The REMOVE Procedure

```
REMOVE(β, descend_set)
1   status[s_β] ← UE; status[f_β] ← UE;
2   remove s_β and f_β from IE and IU sets;
3   for each node γ ∈ descend_set do;
4       remove s_γ and f_γ from V_α;
5       if (s_{α_plan}, s_γ), (s_{α_plan}, f_γ) ∈ E_α remove it from E_α;
6       remove all the edges (s_γ, a) ∈ E_α from E_α;
7       remove s_γ and f_γ from IE and IU sets;
8   UPDATE_IV_SETS(s_β);
```

Figure 24: The procedure for removing recipes from Gr_{α}.