Integration of a PDDL Planning Framework into the Multi-Agent Simulation System LAMPSys

Daniel Höller and Christoph Mies

Fraunhofer-Institute Intelligent Analysis and Information Systems IAIS
Schloss Birlinghoven, D-53754 Sankt Augustin, Germany
{daniel.hoeller,christoph.mies}@iais.fraunhofer.de

Abstract. LAMPSys is a general purpose multi-agent simulation platform that has successfully been used to simulate real-world domains during the last years. A hierarchical agent organization encapsulates decision making by planning centrally for a group of agents. This work introduces a specification framework for the generation of PDDL planning instances from a leader agent's world model. The modeling of the generation process simplifies the integration of different planning domains into LAMPSys, since general operators and specifications can be reused. Another contribution is a plan post-processing that supports plan monitoring. The temporal synchronization of actions in PDDL plans with absolute points in time is extended by a qualitative synchronization with a partial order of actions. In some situations, this approach improves the parallelism of a plan significantly. Finally, the application of PDDL planning to the real-world domain of medical evacuation is evaluated.

Keywords: multi-agent system, simulation, planning, PDDL

1 Introduction

LAMPSys is a general purpose multi-agent simulation platform that has been used to implement decision support systems. During the past years several domains have been modeled, e.g. for business processes or the military. The whole world is modeled in terms of agents, so every change in the world is caused by agent behavior. The domain is defined by a set of agents that is called the *society of agents*. Each agent has a set of attributes that represents the agents internal state and an action function that controls the agent behavior. The principle of autonomy of agents forbids direct manipulation of internal agent state or behavior of other agents, so all interactions between agents are handled via messages. In early LAMPSys versions, complex decision making of agents had to be hardwired, which made it difficult and time-consuming to implement new domains that include complex behavior patterns. Thus, [1] introduces a framework that makes it possible to organize the agents in a hierarchical manner. This simplifies the decision making and is quite similar to structures in business companies and the military. The so-called *leader agents* have the ability to make complex decisions, which is very important for the significance of the whole simulation.

An example for such an agent is the dispatcher agent for the domain of medical evacuation introduced in section 3.1. Each leader agent has a set of subordinate agents, which are assumed to follow orders, i.e. to execute assigned agent actions, and to inform the leader agent about the execution state of the given orders. The tasks and goals of a leader agent are fulfilled by task decomposition to lower level tasks assigned to its subordinates. This recursive process continues until all tasks are solved. A comprehensive description of the whole LAMPSys system can be found in [2]. The design of agents that perform decision making in complex domains remains a difficult task. In order to simplify this process, the elaborated decision making technique *automated planning* [3] has been integrated into LAMPSys. A widely-used language to describe planning problems is the Planning Domain Definition Language (PDDL). Since the first definition as language for the problem domains in the International Planning Competition in 1998, PDDL has been extended to support many features that make it possible to model real-world domains. A detailed introduction to PDDL can be found in [4–7]. The biannual IPC competitions resulted in a growing PDDL community and in open-source planning systems with rising capabilities and performance.

Many real-world domains, and thus many domains in the field of multi-agent simulation, are inherently parallel. The execution of agent behavior consumes time. Parallel and time consuming behavior can be modeled in PDDL by durative actions with discrete or continuous effects [5]. Another useful extension, especially in the evaluation domain introduced in section 3.1, are timed initial literals (TIL) [6], which enable a straight-forward definition of deadlines for some operations. Even though there are not many planning-systems supporting TIL or continuous durative actions available, the recent planning systems are becoming more powerful and can be used to model demanding domains. Section 2 describes a framework that supports the integration of domain independent PDDL planning systems in order to enable the decision making of leader agents. However, the framework can easily be enhanced to support other planning approaches. An evaluation of the approach, especially of the plan post-processing, is discussed in section 3. Finally, section 4 concludes the paper and introduces the next steps of future work.

2 Integration of Planning

The integration of PDDL planning systems into the decision making process of the leader-agents is introduced in the following. The planning behavior is split into three phases. First of all, a *planning instance* is generated in PDDL. Based on this description, a PDDL planning system is able to generate a solution, i.e. a plan, solving the instance. The definition and generation of such a planning instance are described in section 2.1. The resulting plan is post-processed in order to enable a simple execution in LAMPSys. Section 2.2 introduces this post-processing. Finally, the generated plan is executed. The plan monitoring is shortly discussed in section 2.3, but it is not in the main focus of this work.

PDDL Example 2.1 Exemplary PDDL Planning Problem.

```
(define (problem medEvac)
  (:domain logistics1 )
  (:objects
    trans1_pos trans2_pos pack1_pos pack2_pos queue1_pos queue2_pos - position
    trans1 trans2 - transporter pack1 pack2 - package queue1 queue2 - queue
  )
  (: init
    (= ( capacity_trans trans1 ) 3.0)
    (= ( capacity_trans trans2 ) 5.0)
    (at trans1 trans1_pos) (at trans2 trans2_pos)
    (at pack1 pack1_pos) (at pack2 pack2_pos)
    (at queue1 queue1_pos) (at queue2 queue2_pos)
  )
  (:goal
    (and
      ( delivered pack1)
      ( delivered pack2)
    )
  )
)
```

2.1 Generation of Planning Instances

The flexible generation of planning instances is important in order to reduce the effort of integrating new planning domains into the simulation. By providing a specification of the planning instance generation and set of common operators, many elements of the generation can be applied in several domains. A leader agent maintains an internal representation of the world, the *world model*, which is updated in each LAMPSys simulation cycle with the latest perceived sensor updates. The world model especially contains a list of subordinate agents. Each subordinate fulfills a certain *role* in the corresponding decision making task. This role concept encapsulates functional capabilities of agents and abstracts from the concrete agent instance. For example, an agent that is able to transport other agents may fulfill the role "transporter", regardless if it is a helicopter or an ambulance van. When a leader agent has detected a situation that requires planning, the relevant part of its current world model is translated into a planning instance. In PDDL, a planning instance consists of two parts: the planning domain definition and the planning problem. The former contains an abstract definition of the available actions, relations, functions, object types etc. The latter contains a description of the initial state and the goal. The initial state consists of a definition of the available objects, the valid relations, initial values for PDDL functions and so forth. The goal is basically one condition that has to hold after the plan has been executed. [4–7] provide an elaborated discussion of PDDL. The PDDL domain definition is domain-specific and hard-coded. The

planning problem reflects the leader agent's current world state and has to be generated at runtime. The definition that is used to generate the PDDL problem is given by the following tuple:

$$prob = (roles, predicates, functions, goals, metric), \tag{1}$$

which elements are discussed in the following. The agents that are relevant for the planning problem are called *actors*. As already mentioned, each subordinate fulfills a role in the decision making task. The $roles = \{r_0,\ r_1, \ldots, r_n\}$ definition is used to access the relevant agents, i.e. actors. Every $r_i \in roles$ has the following structure:

$$r_i = (rname_i, rfilter_i, \{rattrib_i^1, rattrib_i^2, \ldots, rattrib_i^k\}), \tag{2}$$

where $rname_i$ is the role name that is written as type to the PDDL problem. $rfilter_i$ is a filter-rule that is used to decide which agents fulfill the role. In most cases $rfilter_i$ references directly to the roles in the world-model (e.g. all subordinates that fulfill the role "transporter" are queried), but they do not have to. An example of an actor role that refers to non-subordinate agents is the role "package" denoting injured in the evaluation domain (introduced in section 3.1). In the simulation, the injured call for help and the logistic dispatcher has the task to evacuate them. A special filter operator extracts injured from the world model. The generated PDDL problem definition contains all agents matching $rfilter_i$ as PDDL objects of type $rname_i$. The PDDL example 2.1 contains all actors fulfilling the roles "transporter", "package" and "queue". Additionally, every r_i contains a number of role attribute definitions

$$rattrib_i^j = (raname_i^j, radef_i^j), \tag{3}$$

that include a name $raname_i^j$ and a definition of the calculation of its value $radef_i^j$. These role attributes can be referenced when the predicates and functions are generated (see below). They can either be a reference to an agent attribute, e.g the position of a transporter. More complex definitions requiring calculations based on several attributes can also be formulated. A transporter's available capacity is an example for such a complex calculation, because it is the difference between the overall capacity and the number of transported injured. The generated PDDL problem definition contains for each r_i all attributes $rattrib_i^j$ as PDDL objects as can be seen in example 2.1, where all roles require the attribute "position". The definition of predicates, functions and goals of the PDDL problem is based on these role attributes. The definition of a PDDL relation or function p_i is given by a tuple

$$p_i = (pname_i, (parg_i^0, \ldots, parg_i^l), \otimes, res), \tag{4}$$

where $pname_i$ is the name of the predicate, i.e. function or relation. The argument specifications $parg_i^j$ are used to access role attributes defined above. The result of collecting the values specified in $parg_i^j$ is the tuple $argval_i^j$. Its order is

determined by the order in which the values are stored in the world model of the leader agent. For example, the tuple POS_{trans} of positions of all agents fulfilling the role "transporter" can be specified by that formalism. The value tuples of the arguments are combined by the function

$$\otimes : \mathcal{T}(dom(argval_i^0)) \times \ldots \times \mathcal{T}(dom(argval_i^l))) \rightarrow \mathcal{P}(dom(argval_i^0) \times \ldots \times dom(argval_i^l)), \tag{5}$$

where $dom(x)$ denotes the domain of x, $\mathcal{P}(Y)$ the power set of Y and $\mathcal{T}(Z)$ all tuples without duplicate elements that can be generated from the set Z. $\mathcal{T}(Z)$ is basically the power set of Z with a given order. $\otimes$ maps the argument value tuples to one set of combined argument tuples. Thus, every tuple in the resulting set of $\otimes$ contains all arguments for a correct function or relation evaluation, respectively. One important combination is the Cartesian product combination $\otimes_{cart}$, which combines each argument value $v_j \in argval_i^j$ with all other values $v_k \in argval_i^k$ for all $k \neq j$. Another important combination is $\otimes_{group}$, which combines two tuples with same length $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$ by their indices: $\otimes_{group}(A, B) \mapsto \{(a_1, b_1), \ldots, (a_n, b_n)\}$.

Assuming that $AGENTS_{trans}$ is the tuple of all agents fulfilling the role "transporter" and the tuple POS_{trans} contains the corresponding agent positions, $\otimes_{group}(AGENTS_{trans}, POS_{trans})$ generates a set of tuples such that each agent is assigned its own position. This combination can, for example, be used to model the relation $At \subseteq AGENTS_{trans} \times POS_{trans}$ stating at which position a transporter is located. Each combination of argument values is passed to the result function $res : dom(argval_i^0) \times \ldots \times dom(argval_i^l) \rightarrow \{true, false\} \cup \mathbb{R}$, which calculates the result of that argument value combination. When p_i is a relation definition, res computes either $true$ or $false$ stating whether the given arguments are in that specific relation p_i or not. If p_i denotes a function definition, res calculates the function value $v \in \mathbb{R}$ of the argument tuple. The function "capacity_trans" and the relations "delivered" as well as "at" are contained in the example problem 2.1. The goal statement is a conjunction of relation entries, but can easily be extended for other logical connectors or arithmetic relations, e.g. $<$. Relations and functions are specified as defined above. The formula of the optional metric can recursively be defined by arithmetic operators.

The whole process of the planning problem generation is summarized in the following. First of all, the actors are identified by means of their roles in the decision making task. Depending on the role, certain role attributes are extracted or calculated, respectively. These role attributes can be used as arguments in functions and relations. The combination of the different attributes is determined by a combination function $\otimes$. After the generation of the argument value tuples, the result function res computes the resulting function value or determines whether the relation holds, respectively. The goal is based on relations as well as functions and can be specified analogously. After the planning problem has been generated, the planning system is called. Since the systems usually participate at the IPC, a common system call is available. The resulting plan is parsed and post-processed as described in section 2.2.

2.2 Plan Post-processing

The generated PDDL plan has to be translated and executed in the simulation system. As mentioned in the introduction, parallelism and time consuming behavior are modeled in PDDL with durative actions. The time synchronization is done by absolute starting times for the action execution. In real-world domains an exact prediction of the execution time of actions is hardly achievable, since expected duration usually can only be estimated. Thus, the quantitative time synchronization provided by PDDL is a challenge for the plan execution and monitoring. One solution is a qualitative time synchronization, which requires a predecessor relation between the actions of a PDDL plan. If this relation considers the dependencies between the actions correctly, a simple plan execution becomes possible: "always execute the actions which predecessor actions have been executed completely". In order to establish such a predecessor relation, a set of ordering constraints $\prec$ is introduced. This approach is similar to the ordering constraints used in partial ordered plans defined in [3, Definition 5.1]. When the following two conditions hold, $\prec$ considers the dependencies between the actions inside the plan correctly:

1. All preconditions of an action a have to be established by the execution of its preceding actions.
2. All action executions that potentially disturb a successful execution of a are either predecessors or successors of a.

The information about the dependencies of the different PDDL actions is implicitly given in the PDDL domain description. The exploitation of this knowledge for the detection of interfering actions enables a generic approach for all PDDL domains, reduces redundancy and minimizes effort for the domain designer. Fox and Long provide the following rules for the detection of two interfering actions as well as the definitions of the used terminologies [5, ch. 7]. Two actions a and b of a PDDL plan π are non-interfering iff the equations 6 to 9 hold:

$$GPre_a \cap (Add_b \cup Del_b) = GPre_b \cap (Add_a \cup Del_a) = \emptyset \tag{6}$$

$$Add_a \cap Del_b = Add_b \cap Del_a = \emptyset \tag{7}$$

$$L_a \cap R_b = R_a \cap L_b = \emptyset \tag{8}$$

$$L_a \cap L_b \subseteq L_a^* \cup L_b^*, \tag{9}$$

where $GPre_a$ is the set of atoms that appear in the propositional precondition of the action a. An atom or grounded atom of a planning instance is a fully instantiated predicate. The sets Add_a and Del_a contain the positive and negative ground atoms that are contained in the effects of a. Numeric effects consist of an assignment operator, e.g. `assign` or `increase`, followed by two numeric expressions referenced as $lvalue$ and $rvalue$. The sets L, R and L^* contain primitive numeric expressions (referenced as f): $f \in L_a$ iff f appears as an $lvalue$ in the action a, $f \in R_a$ iff f appears as an $rvalue$ in the action a or is used in its precondition. $f \in L_a^*$ iff f appears as an $lvalue$ in an additive assignment effect in a. More details can be found in [5, ch. 7]. Based on these definitions,

an algorithm can be defined that transforms the PDDL plan $\pi = \{act_1, ..., act_n\}$ generated by the planning system into a partial ordered plan. Each act_i is a pair (t_i, a_i), a_i is a (grounded) action and t_i is a positive real value that represents the starting time. The post-processed plan is a tuple $\pi_\prec = (A, \prec)$, where A is the set of all grounded actions that occur in π and $\prec$ a set of ordering constraints of the form $(a \prec b)$, which states that the execution of a has to be finished before the execution of b may be started. Let $I_\pi \subseteq A \times A$ be a relation that contains all interfering grounded actions of π according to the conditions 6 to 9. The set of ordering constraints $\prec = \prec_1 \cup \prec_2$ is defined by the two conditions given in 10 and 11.

$$\forall (t_a, a), (t_b, b) \in \pi : (I_\pi(a, b) \wedge t_a < t_b) \Leftrightarrow (a \prec b) \in \prec_1 \tag{10}$$

Two non interfering actions a and b can be executed simultaneously. When they are interfering, i.e. $I_\pi(a, b)$ holds, the order of the starting times in π determines their relationship in $\prec$. Simultaneously started actions cause a problem, because if $t_a = t_b$, $a \prec b$ and $b \prec a$ hold at the same time. Note that this case should occur rarely, because the execution of two interfering actions should not happen simultaneously. However, condition 11 provides a solution for this special case by introducing only one (randomly chosen) ordering constraint if $t_a = t_b$:

$$\forall (t, a), (t, b) \in \pi : (I_\pi(a, b) \wedge a \neq b) \Leftrightarrow ((a \prec b) \in \prec_2) \underline{\vee} ((b \prec a) \in \prec_2) \tag{11}$$

where $\underline{\vee}$ denotes an exclusive disjunction. The described post-processing is primarily intended to enable a simple and flexible execution of temporal plans consisting of durative actions, which are used in the experiments described in section 3. In principle, it would also be possible to use it for parallelization of sequential plans. Thereby note the side effect that an action in $\pi_\prec$ is always executed as soon as possible. This is no restriction and even an optimization for the medical evacuation domain introduced in section 3.1, but might cause problems in other domains, e.g. when opening hours or deadlines have to be considered. Thus, the post-processing has to be adapted to support planning instances that include timed initial literals.

2.3 Plan Monitoring

The new plan representation allows a very simple plan-execution that starts all actions which predecessors according to $\prec$ have already been finished. Let S and F be the set of all actions, which execution has been started or finished, respectively. The execution of the actions in N is started in the current cycle:

$$N = \{a \in A | a \notin S \cup F, \forall b \in A : (b \prec a) \rightarrow (b \in F)\} \tag{12}$$

After the execution of an action a has been started, a is added to S and translated into an order by the leader agent. The order is send as message to the corresponding subordinate *sub*. During the execution, *sub* sends discrete status notifications to its leader agent, which can be used for plan monitoring. Beside standard notifications in case of execution failure or success, intermediate states

can also be communicated. When an action has been finished, it is moved from S to F. The leader agent monitors the plan execution based on the discrete status notifications of its subordinates and other changes in its perceived world.

3 Evaluation

The performance of the generated PDDL planning instances and the introduced plan post-processing in the evaluation domain medical evacuation (MedEvac) is discussed in the following. MedEvac is a real-world domain, where a parallel action execution is vital. It is introduced in section 3.1. The setup of the evaluation experiments and its results are discussed in sections 3.2 and 3.3, respectively.

3.1 Medical Evacuation Domain

The MedEvac domain is a logistics environment that simulates the evacuation of injured persons. The rescue operation is planned and controlled by a single leader agent, the *logistic dispatcher*. It watches out for help-calls of new injured and commands a number of subordinates, which are rescue helicopters, ambulance vans or hospitals. The transporters can have different average speeds, cruising ranges or transport capacities. When the cruising range of a transporter has been exhausted, it can be refilled at a garage. During the simulation, agents representing injured persons are placed randomly in a certain area. They call for help and thereby trigger a rescue operation. The dispatcher agent commands its subordinate transporters to evacuate the injured to a hospital and orders the hospital to perform a medical care. Depending on the degree of injury, an injured is assigned a time limit within it must be evacuated to a hospital. This can easily be modeled with timed initial literals being supported by PDDL 2.2 [6]. However, not many planning systems support this extension, so the experiments here do not make use of it. The MedEvac domain is fully observable and deterministic, but can easily be extended to become partial-observable or non-deterministic. Except for the occurrence of new injured, the domain is also Markovian. A more elaborated introduction of the domain can be found in [1, 2].

3.2 Experiment Setup

Many domains modeled in LAMPSys represent real-world scenarios and are therefore inherently parallel. The hierarchical decision model causes one leader agent to plan for its subordinates. Thus, a central planning mechanism results in a plan that is executed distributedly by several agents. Hence it is essential to have a good parallelization to make use of all given resources. So the experiments are designed to show how well a planner parallelizes its solutions. Another evaluation criterion is the problem size that current PDDL planning systems can cope with. Therefore the experiments are designed in the following way: at simulation start (and only then) a number of agents is placed randomly in a certain area of the environment. The following agent types are used: injured,

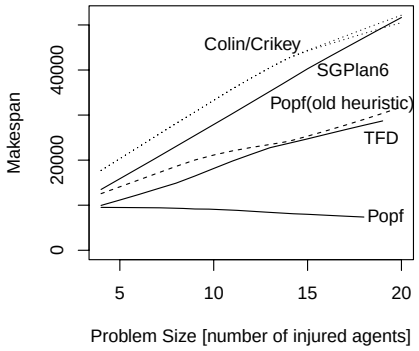

Fig. 1. Makespan vs. Problem Size

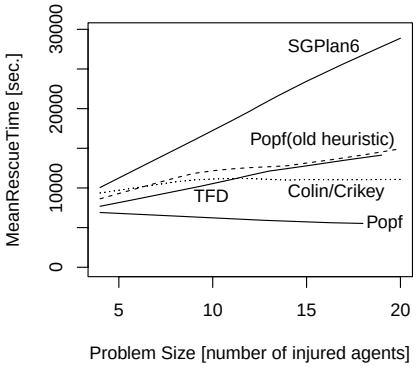

Fig. 2. MeanRescueTime vs. Problem Size

helicopters, hospitals and garages to refill the helicopters. During the experiment series, the number of each agent type is increased in a given interval, so that the ratio between the agent types stays nearly the same. Thus, a transporter has to evacuate roughly the same number of injured. Analogously, each hospital has to treat a similar number of injured in each run. The overall time t of the rescue operation is supposed to stay constant if the plan is well parallelized. In the PDDL plans, t is the *makespan*. In LAMPSys, t is the point in time at which the last injured is rescued. It is also called the *maximal rescue time (maxRescue-Time)*. All planning systems have to solve exactly the same planning problems and must not exceed a time limit of 5 minutes for each calculation. They have to optimize the makespan of the plans.

3.3 Results

Figure 1 shows the makespans of the generated PDDL plans depending on the problem size, which is measured by the number of injured in the domain. It has been smoothed using "locally weighted scatterplot smoothing" (LOWESS). Even in small planning problems, the makespans of the plans show rather large differences between the different planning systems. The scaling behavior with increasing problem sizes differs also. POPF [8] manages to generate the shortest makespans and scales well with increasing problem size. It is the only system that achieves constant makespans on all problem sizes. POPF with the old heuristic generates plans with larger makespans, which are still rather good compared with the other planning systems. The TFD [9] system solves small problems nearly as well as POPF, but the makespans increase with larger problem sizes; here it is on one level with POPF using its old heuristic. The makespans of the other planning systems do not scale very well. Colin [10] and Crikey [11] generate nearly the same results.

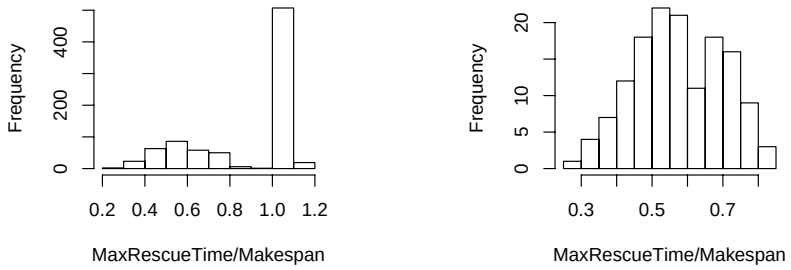

Fig. 3. Histograms of $\frac{maxRescueTime}{makespan}$: All planning systems (left) vs. Colin (right)

The percentage of solved problems by TFD and POPF systems decrease when 16 injured are in the problem. They are able to solve approximately 80% of the problems with 17 injured. No plans are found within the time limit when there are 19 (POPF) or 20 (TFD) injured available. Crikey solved 80% of the problems with 20 injured, Colin and POPF with the old heuristic still approximately 70%. SGPlan [12] solved all problems including 20 injured.

In the MedEvac domain the main quality criterion is the time that is needed to rescue the injured. Therefore figure 2 shows the *average rescue time (meanRescueTime)* measured in the simulation, i.e. during plan execution, depending on the problem size. The diagram has also been smoothed using LOWESS. The curves of all systems except Colin and Crikey are similar to the curves of figure 1, which confirms the expectation of a correlation between planned makespan and expected rescue time during plan execution. But the plan execution of Colin and Crikey plans performs unexpectedly good. This might have several reasons: (1) there is no correlation between the makespan and the meanRescueTime which would be a problem for the application of PDDL in MedEvac, because the planning systems can not optimize the meanRescueTime, or (2) the plans from Colin and Crikey are improved by the post-processing. To evaluate the influence of the post-processing on the maximal rescue time, the makespans have to be compared with maxRescueTime. Figure 3 depicts this relationship by providing the distribution of $\frac{maxRescueTime}{makespan}$. The diagram on the left side shows the distribution over all planning systems, which has two peaks: one in the interval $[1.0; 1.1]$ and a smaller one in $[0.5; 0.6]$. The right histogram shows the distribution only for the results of Colin (Crikey looks rather similar). It is distributed around a mean value of 0.57, with minimum and maximum values of 0.28 and 0.82. So there is a situation where the actual execution of the plan needs only 28% of the planned time. When the results of Colin and Crikey are not included, the distribution has the supposed shape: there is a linear relationship between maxRescueTime and makespan with a correlation coefficient of 1.00. This is important, because it shows that the actions are executed in the planned way. The mean value of

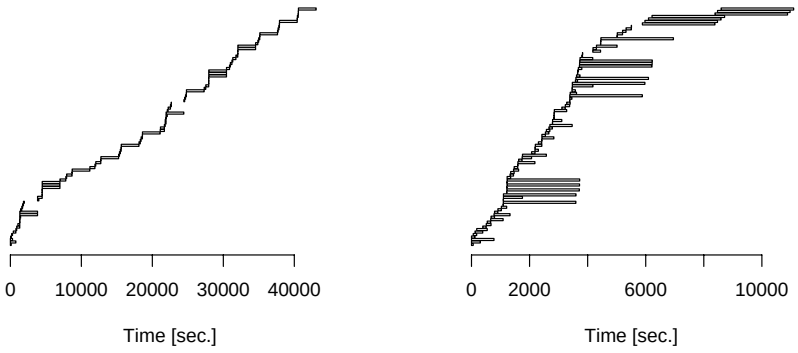

Fig. 4. Gantt charts of a Colin plan before (left) and after (right) post-processing

this relation is 1.04, i.e. the plan execution needs 4% more time than planned. This is caused by the duration of agent communication in LAMPSys.

The improvement caused by the plan post-processing generated by Colin increases when the problem size rises. The effects of the post-processing are illustrated in figure 4. It shows Gantt diagrams of a plan of Colin before and after the post-processing. The actions of the original solution are executed rather sequential but can be parallelized by the post-processing. The plans of Colin and Crikey make use of non-interfering grounded actions and can significantly be improved in the post-processing step. Contrary to this the plans of the other planners can not be improved because their actions are interfering.

4 Conclusion

The integration of PDDL planning into LAMPSys introduced in section 2 simplifies the implementation of decision making behavior of leader agents. The specification framework for the generation of planning instances in PDDL from LAMPSys enables the domain designer to model the generation process itself. Thus, new planning domains can be integrated with less effort, since general operators and specifications can be reused. The framework can be extended to support other planning approaches. The post-processing of PDDL plans reduces the effort of plan monitoring and execution. Additionally, it can be used to parallelize sequential plans. The basic idea is to use a qualitative time synchronization with a partial order instead of using absolute synchronization times. Note that this approach is not valid for all domains since every action is started as early as possible. This may cause problems when using timed initial literals, e.g. when opening hours are violated. However, it results in an optimal behavior for MedEvac. In section 3, the evaluation domain MedEvac is introduced as

real-world domain. It is vital to execute parallel plans in a distributed manner in order to perform optimally. The evaluation shows that there are PDDL planning systems that perform well in planning central for distributed plan execution. The best performing planning system shows a good scaling when the planning instances become larger. The evaluation reveals a linear correlation between the plan makespans of most of the planning systems and their actual execution time in the simulation, i.e. the actions are executed as planned by the planning systems. The solutions of two planning systems have even been improved by the post-processing through further parallelization, but this highly depends on the concrete plan and planning system.

Future work comprises a modification of the post-processing in order to avoid too early action execution starting times when timed initial literals are in the PDDL planning instance. Additionally, the performance of the PDDL planning systems in MedEvac will be compared with domain-configurable systems like HTN or domain-specific solutions.

References

1. Mies, C., Mertens, R.: Integration of a Hierarchical Decision Making Framework into the Multi-Agent Simulation System LAMPSys. KI 2009. 32nd Annual Conference on Artificial Intelligence. Workshop Proceedings (Online) (2009) 273–284
2. Mies, C., Wolters, B., Steffens, T.: LAMPSys - An experiment-platform for the optimization of processes and agent-behaviors. KI Zeitschrift 2 (2009) 42–45
3. Ghallab, B., Nau, D., Traverso, P.: Automated Planning - Theory and Practice (2004), Morgan Kaufmann Publishers Inc., San Francisco, CA, USA
4. McDermott, D.: The 1998 AI Planning Systems Competition. AI Magazine (2000) (vol. 21)
5. Fox, M., Long, D.: PDDL2.1: An extension to PDDL for expressing temporal planning domains. Journal of Artificial Intelligence Research (2003) (vol. 20)
6. Edelkamp, S., Hoffmann, J.: PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition. Proceedings of the 4th International Planning Competition (2003)
7. Gerevini, A., Long, D.: Plan Constraints and Preferences in PDDL3. Technical Report RT 2005-08-47, Dept. of Electronics for Automation, University of Brescia (2005)
8. Coles, A. J., Coles, A. I., Fox, M., Long, D.: Forward-Chaining Partial-Order Planning. In Proceedings of the 20th ICAPS (2010)
9. Eyerich, P., Mattmüller, R., Röger G.: Using the Context-enhanced Additive Heuristic for Temporal and Numeric Planning. In Proceedings of the 19th ICAPS (2009)
10. Coles, A. J., Coles, A. I., Fox, M., Long, D.: Temporal Planning in Domains with Linear Processes. 22th IJCAI, (2009)
11. Coles, A. I., Fox, M., Long, D., Smith, A. J.: Planning with Problems Requiring Temporal Coordination. Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 08) (2008)
12. Hsu, C. W., Wah, B. W.: The SGPlan Planning System in IPC-6. In Proceedings of the 19th ICAPS (2009)