Decomposition of planning problems

Article *in* AI Communications · January 2006

Source: DBLP

CITATIONS
31

READS
624

3 authors, including:

Laura Sebastia
Universitat Politècnica de València
72 PUBLICATIONS **1,274** CITATIONS
SEE PROFILE

Eva Onaindia
Universitat Politècnica de València
183 PUBLICATIONS **2,060** CITATIONS
SEE PROFILE

Decomposition of planning problems

Laura Sebastia, Eva Onaindia and
Eliseo Marzal
Technical University of Valencia,
Camino de Vera s/n, 46022, Valencia, Spain
E-mail: {lstarin, onaindia, emarzal}@dsic.upv.es

The ability to decompose a problem into manageable sub-components is a necessity in complex problem-solving tasks. In planning, the application of a divide-and-conquer methodology is known as *planning decomposition*. This technique consists of the following: decomposing a problem into smaller components (subproblems), solving these subproblems individually, and then combining the obtained solutions. The success of this technique is subject to the interactions that may appear between actions from solutions for different subproblems.

In this paper, we present a novel technique, STeLLa, to overcome the inherent difficulties in planning decomposition. This technique partitions a planning problem in such a way that its subproblems can then be solved separately (either sequentially or concurrently) and their solutions can be easily combined. The key issue is that interactions among goals are used to come up with the problem decomposition rather than solving them once the problem is decomposed. This approach proves to be very beneficial with respect to other decomposition methods and state-of-the-art planners.

Keywords: planning, problem decomposition

1. Introduction

Classical AI Planning has experienced great advances over the last few years as search strategies become more and more efficient. Techniques like graph analysis ([4],[33]), local search ([1],[20]) or heuristic search ([5],[23],[38]) are aimed at focusing only on the most relevant parts of the search space for the planning problem.

Separating a problem into different parts (*divide-and-conquer* methodology) has been widely used in logic and mathematical proofs, modular program design, etc. The field of AI planning also has its counterpart, known as *planning decomposition*. This technique consists of the following steps: decomposing a problem into smaller components (subproblems), solving these subproblems individually, and then combining the obtained solutions. Planning decomposition offers several advantanges such as:

1) The obtained subproblems are much simpler than the original problem and are therefore easier to solve. In general, the process of exploring the search space of all individual subproblems is much less costly than exploring the whole space of the original problem.
2) It is not necessary to design new planning approaches to solve a decomposed problem; the same techniques used to solve a complete problem can be re-used to solve the subproblems that result from the decomposition.

Planning decomposition relies on three fundaments: a) the principles by which to divide the problem into different subproblems, b) the resolution process used to solve each subproblem individually and c) the technique used to combine the obtained solutions. The first item is the crucial issue as it determines the efficiency of the resolution process and the complexity of the technique for plan combination. This complexity is greatly determined by the interactions that may emerge between actions from the solutions for different subproblems. We can distinguish between **negative interactions** (when achieving a fact deletes other necessary facts) and **positive interactions** (when one action helps achieve several necessary facts).

It is clear that negative interactions among the different subplans involve a great complexity in the process of plan combination. Conflicts must be detected and repaired, and this task may make the plan combination process more costly than solving the original problem itself. Therefore, it is important to come up with a decomposition technique that addresses this aspect. Positive interactions directly affect the solution quality. Although each subproblem aims at solving its particular goal(s)

AI Communications
ISSN 0921-7126, IOS Press.

depending on the decomposition technique used, parts of the subplans might overlap; i.e., the same set of actions may solve the same subgoals in different subplans. This is also an important aspect that must be considered in order to avoid repeating the same computation.

Planning decomposition has not been widely used to solve planning problems due to the difficulty of finding a "good" decomposition that leads to obtaining a better performance with respect to solving a non-partitioned problem. As stated above, it is important to choose a problem partition that involves the lowest possible number of conflicts and takes into account the positive interactions among the actions of different subplans. This division can be provided by the domain engineer (as in HTN planning -[8],[36],[42]-), or an automatic process to obtain a problem decomposition can be used [44]. The latter requires an analysis of the domain and problem structure in order to identify which parts of the problem are independent (to a certain degree) from each other. This brings us again to the fact that the decomposition method must provide efficient partition criteria in order to achieve a low-cost process for plan combination. It is also important to note that when planning decomposition is used, the global view of the problem is lost, and this may lead to a degradation of the plan quality.

However, one of the most important gains from using an appropriate decomposition technique is the possibility of solving subproblems concurrently. If subproblems are fairly independent; i.e., the resolution of a subproblem does not depend on the result of executing other subproblems, we can then consider a concurrent resolution rather than a sequential one, which may result in very great time savings.

In this paper, we present a new technique to partition planning problems aimed at mitigating the inherent shortcomings in planning decomposition. Our technique is able to automatically extract the different subproblems easily. These subproblems are independent enough so that 1) we can apply a concurrent resolution, 2) we can combine the solutions in such a way that the number of conflicts is very low and 3) we can use any existing planner to solve the subproblems. At the same time, our technique achieves plans with a similar (or even better) quality than the solution to the original problem.

This paper is organized as follows. Section 2 briefly describes some other decomposition techniques. Section 3 introduces the process for decomposing a planning problem and how the solutions are combined when the obtained subproblems are solved sequentially. Section 4 explains how these subproblems can be solved concurrently. Both Section 3 and Section 4 present some experiments that compare our decomposition technique with other planning systems that use a similar decomposition approach. They also show how our decomposition technique works when using different planners to solve the obtained subproblems. Finally, Section 5 summarizes the decomposition technique introduced in this paper and highlights future work.

2. Related work

In this section, we summarize some previous decomposition approaches in the literature. First, we introduce some notations. A *state* s is a set of logical facts. A *planning problem* $P = \langle \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ is a triple where:

- $\mathcal{A}$ is the set of actions that can be applied in the domain. Each *action* a is a triple $a = (pre(a), add(a), del(a))$, where $pre(a)$ is the set of action's preconditions, $add(a)$ is its *add list*, and $del(a)$ is its *delete list*, each of which constitutes a set of facts.
- $\mathcal{I}$ (initial state) and $\mathcal{G}$ (goal state) are sets of facts.

Given an action a and a state s, a is said to be applicable in s when $pre(a) \in s$ and the result of applying a to s is:

$$Result(s, a) = s \cup add(a) - del(a)$$

A *plan* is an action sequence $\mathcal{P} = \langle a_1, a_2, \ldots, a_n \rangle \in A^*$. In case $\mathcal{P}$ is applicable in the initial state $\mathcal{I}$ of a planning problem, it defines the following resulting state:

$$Result(\mathcal{I}, \mathcal{P}) = Result(Result(\ldots(Result(Result(\mathcal{I}, a_1), a_2), \ldots), a_{n-1}), a_n)$$

A *solution plan* for a problem $P = \langle \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ is a plan $\mathcal{P}$ such that $\mathcal{G} \in Result(\mathcal{I}, \mathcal{P})$.

The remainder of this section describes the existing decomposition approaches, which can be classified according to two features:

- **Method of decomposition**:

  - **automatic decomposition**. This technique obtains information by analysing the domain and the problem structure and uses this information to achieve a problem decomposition.
  - **manual decomposition**. This is the type of decomposition used when an automatic decomposition is difficult to apply (for example, in aerospace applications [9], military systems [13], etc.). This problem decomposition is performed by the domain engineer, who mainly studies the interaction between the different subproblems.

- **Type of resolution**:
  - **sequential resolution**. The resolution of a subproblem depends on the resolution of other subproblems. Therefore, each subproblem is solved after all those it depends on are solved.
  - **concurrent resolution (including multiagent planning)**. In this case, we find a distributed planning environment that can be defined from two different points of view:

    1. A planning system is said to be distributed when the subproblems obtained from a decomposition are independent enough to be solved concurrently.
    2. Distributed planning is the problem of finding a plan that helps a set of agents in a given initial configuration to collectively satisfy their goals [35].

    By using distributed planning, we can benefit from the efficiency of parallel processing and the robustness of distributed systems. However, only problems with any of the following features can be solved by distributed planning:

    - inherent distributed problems, for example, those located in different places or distributed due to privacy and security reasons
    - problems that can be decomposed into subproblems with limited interactions
    - problems where different agents need an independent but coordinated plan, etc.

In accordance with the above classification, Table 1 shows some planning approaches that follow a resolution scheme based on problem decomposition. Some of these techniques are explained in detail below. We focus on those approaches that are based on an automatic decomposition with a sequential or concurrent type of resolution, except those that have been developed to work in multiagent environments, which are briefly summarized in Table 2.

The **traditional view** in problem decomposition consists in partitioning a goal set $\mathcal{G}$ in disjoint subsets $\mathcal{G}_i$, such that $\mathcal{G} = \bigcup \mathcal{G}_i$. In this approach, presented in [44], a plan $\mathcal{P}_i$ to solve each $P_i = \langle \mathcal{A}, \mathcal{I}, \mathcal{G}_i \rangle$ is computed, and then plans are reassembled for an overall solution. In Korf's analysis [31], he discusses computational complexity issues that are related to decomposing a compound goal into subgoals in forward-chaining, total-order planners. His conclusion is that, in the case that all subgoals are completely independent of each other, decomposition cuts down both the branching factor and the depth of the search, leading to an exponential amount of savings in planning time. Unfortunately, this is not always the case. Therefore, in the process of combining plans, hard conflicts among subplans may arise, which are often caused by actions that share the same resources.

Following this traditional approach, Iwen and Mali ([25], [26]) developed a technique that builds interaction graphs to partition the goal set $\mathcal{G}$. The idea is to link literals in the initial situation, $\mathcal{I}$, and the goal set, $\mathcal{G}$, if they refer to the same object in the domain. This draws a bipartite graph, called an *interaction graph*. Then, $\mathcal{G}$ is divided into two sets, where each set corresponds to a group of connected components in the interaction graph. The two obtained subproblems are then solved by using any planner (in [25], FF, Graphplan, and HSP are used). The plans obtained are then combined and conflicts that may arise are solved to obtain the solution for the original problem. The experiments indicate important benefits with respect to the execution time. The most important difference with the traditional approach is that, in this case, the problem is always divided into two subproblems.

A different type of problem solving is **planning with abstraction hierarchies**. Under this approach, a domain is broken down into several levels on a hierarchy. The most critical part of a planning problem is represented at the highest level, and a solution obtained at the lowest or concrete level is a solution to the original problem. The abstraction paradigm was first used in ABSTRIPS [39] and was

Type of decomposition	Resolution	
	Sequential	Concurrent/Multiagent
Automatic	Abstraction hierarchies	Traditional
	Subproblem type	Disjunctive planning
	Goals ordering	Partial global planning
	Subgoals ordering	Local planning
User decomposition	HTN	Distributed HTN

Table 1
Planning systems using problem decomposition

	Description	References and applications
Disjunctive planning	The search space is partitioned and each agent explores a part of this space and works on the complete plan.	[27]
Partial global planning	Each agent mantains a partial global plan, where it stores its own partial view of other agents' plans. Each agent's plan contains a set of goals, a long-term strategy, a short-term strategy, and a set of predictions and statistics to compute heuristics to guide the search.	[14], [10] Cooperative information [11] Cooperative robots [6]
Local planning	Problem decomposition is based on local interaction regions so that each region is assigned to a different agent, which computes its plan independently and communicates to the other agents to avoid interactions. An interaction region is a set of requirements and a set of operators that are related in some way.	Collage system [32]
Distributed HTN planning	The user specifies which agent is in charge of solving each goal. Each agent solves its goals by applying a HTN algorithm. Plans representation is extended to include a node that represents a call to another agent's plan and some primitive actions to synchronize multiagent execution.	Distributed NOAH [7] Distributed SIPE-2 [12] NASA systems [9] Military applications [13]

Table 2
Short description of decomposition techniques in a multiagent environment

later extended in Knoblock's ALPINE [28] and Bacchus and Yang's HIPOINT [3]. Despite the similarity in the general framework, major differences remain between *abstraction hierarchies* and the *traditional problem-domain decomposition*. First, in decomposition planning, the planning process occurs concurrently instead of sequentially as dictated by an abstraction hierarchy. A second difference, which was pointed out by Lansky [32], is that in obtaining the abstraction hierarchy used by ALPINE and HIPOINT, a requirement is enforced such that no interferences between operators at different levels of abstraction exist. This requirement is relaxed in decomposition planning as in-

teractions between groups of goals are allowed.

In large complex planning problems, it is likely to identify different types of subproblems. A type of decomposition for these problems is based on the concept of using **specialised solvers for each subproblem type** as it is more efficient to use different planning techniques depending on the type of subproblem to solve. The motivation for this is that many hard specialised problems have been the subject of research themselves and good solutions exist for them. In contrast, while a generic problem-solving technology is very unlikely to challenge these specialised solutions when they are applied to these problems. These specialised problems often appear as subproblems within a larger planning problem. For example, a problem that involves transporting components between locations, constructing complex artifacts with the components, and then delivering them to customers can contain the following subproblems: route-planning, resource allocation, job-shop scheduling, and construction planning. Each of these can, to some extent (although not entirely) be decoupled from the others and solved using specialised technology.

The REALPLAN [40] planner extracts the resource-scheduling subproblem to be solved by a specialised solver. However, in this case, the task of identifying the subproblems relies on the domain-engineer, who must understand the characteristics of the subproblems and the restrictions on the capabilities of the specialised solvers for the decomposition and the combination of the obtained subplans to be successful. Unlike REALPLAN, Hybrid STAN ([34], [16]) does decompose the problem according to the subproblem type automatically. It involves the automatic identification of a set of subproblems, and the integration of specialized solvers, one for each identified subproblem, with a planner in the solution of the complete problem. A domain analysis tool, such as TIM [15], can be used to decompose the problem and identify the presence of such subproblems.

The main advantage of this approach is that each subproblem can be solved more efficiently; however, the identification of subproblems is complex for each specific case because slight changes in the domain may hinder the subproblem detection. Another inconvenience is the integration between the planner and the set of specific solvers. This technique has produced some limited benefits related to specific domains.

Although planning decomposition in its traditional view has not been widely exploited, some of its underlying principles are used in many planning algorithms. For instance, Koehler and Hoffmann [29] introduced the technique called GAM (*Goal Agenda Manager*), which performs a **goal ordering**. This technique basically consists of ordering the literals of the top-level goal according to the notion of *reasonable ordering*. This notion states that a pair of goals A and B can be ordered so that B is achieved before A if it is not possible to reach a state in which A and B are both true from a state in which just A is true without having to temporarily destroy A. In such a situation, it is reasonable to achieve B before A to avoid unnecessary effort. Specifically, a total order is established between incremental subsets of literals of the top-level goal $\mathcal{G}$. This way, the problem is decomposed in the sense that it is possible to plan from the initial state to the first of these subsets; from the state reached in the previous step to the second subset; and so on until the last subset is solved (which coincides with $\mathcal{G}$). GAM was implemented as a module of the IPP planner [30], obtaining a better performance. However, with this decomposition technique, the planning process cannot occur concurrently because the initial state of a subproblem is the resulting state after solving the previous subproblem (sequential resolution).

More specifically, **subgoal ordering** is a common mechanism to obtain problem decomposition. Subgoal ordering is part of the planning process itself as it helps determine which subgoal (literal) is more convenient to achieve first. Moreover, the orderings among subgoals allow the grouping together of sets of literals that will lead to a problem partition.

Porteous, Hoffmann and Sebastia ([37], [24]) have developed a decomposition technique based on the concept of landmark. A landmark is a literal that must be achieved in any solution plan. These landmarks are then ordered according to three types of orders[1]. As a result, a landmark generation graph (LGG) is obtained. Each subproblem results from considering the leaf nodes of the current LGG, and when a subtask has been processed, the LGG is updated by removing the achieved leaf nodes. The solution plan is formed by concatenating the subplans obtained from each subproblem.

[1] These orders will be formally introduced in Section 3.1.

This technique will be further explained in Section 3.4.2.

GRT [38] decomposes the original problem into subproblems that must be solved sequentially. This decomposition is based on the definition of XOR constraints, which define a relationship between literals. A XOR constraint between two literals is true in a state if only one of them is true in that state. Constraints of this type are called invariants in analysis of domain systems such as TIM [15] and DISCOPlan [19]. For example, in the logistics domain [2], we can define the following XOR relationship: $xor\ (at(x\ *)(in(x\ *))(package\ x)$; that is, a package x can only be in one place at the same time, either in a truck or in an airplane. Once the XOR constraints are computed, it is possible to establish orders between the subgoals that must be reached before satisfying the problem top-level goals. These subgoals are grouped into ordered intermediate goals, which make up the subproblems to solve sequentially.

A recent planner that uses decomposition planning and that has shown an impressive performance in the last planning competition[2] is SGPlan [43]. SGPlan was awarded the first prize as the best planner at the suboptimal metric temporal track and was awarded the second prize in the suboptimal propositional track. It is a good example of how decomposition techniques can play an important role in AI planning. SGPlan partitions a planning problem into subproblems, orders the subproblems according to a sequential resolution of their subgoals, and finds a feasible plan for each goal. Using the extended saddle-point condition [41] and constrained search, new constraints are enforced to ensure that facts and assignments in a later subgoal are consistent with those of earlier subgoals. This technique will be also discussed in Section 3.4.2.

In this section we have described several approaches for decomposing planning problems and how they have been used in different planning algorithms. In summary, we can say that planning decomposition has not been widely used due to the difficulties that arise in combining the obtained subplans when dealing with highly interactive goals. However, the SGPlan planner does make use of this technique as a global problem-solving technique, and had an excellent performance in the last planning competition. In the remainder of the paper, we will present our decomposition technique for planning problems, and we will show that our planner outperforms some of the approaches described above. It also exhibits excellent behaviour thanks to a concurrent resolution of the planning process, among other features.

[2] http://ipc.icaps-conference.org/, 2004

3. Decomposition technique

In this section, we describe our decomposition technique, STeLLa, for STRIPS domains. As it was explained in the previous section, most of the decomposition techniques find difficulties when merging the subplans for an overall solution. Our goal is thus to come up with a decomposition technique that allows the partitioning of a problem into a collection of subproblems whose solutions can be easily combined. In this sense, it is important to analyse the different sources of conflicts that may appear when merging the solutions of the subproblems:

- **Negative interactions**: There is a negative interaction between two actions when the effect of one action is the negation of another action's effect (*inconsistent effects*), or when one action deletes the precondition of another action (*interference*)[3].
- **Positive interactions**: There is a positive interaction between two actions when both need the same precondition and at least one of them does not remove it; that is, the same producer action can be used to support the precondition of both actions.

When combining two subplans, negative interactions may cause conflicts that are repaired by adding new ordering constraints between interfering actions or by adding new actions to reachieve the deleted facts. On the other hand, the combination of two subplans may lead to a plan with redundant actions due to positive interactions.

These interactions make the process of combining subplans difficult. Our objective is to decompose the problem in such a way that these interactions affect the process of plan combination as little as possible. Specifically, we perform a *chrono-*

[3] The terms "inconsistent effects" and "interference" were introduced in [4].

logical decomposition of the problem by considering the *intermediate positions* that each object in the domain must reach throughout the execution of the plan. The subplans obtained for these decomposed problems will be conflict-free and, therefore, the overall plan is achieved by simply concatenating the different subplans. Unlike existing approaches that deal with conflicts at the time of combining subplans, we use the possible interactions to accomplish the plan decomposition. The most important feature of our decomposition technique is that interactions among actions are used as the basic principle for dividing the problem rather than considering them to be a source of conflicts.

In order to illustrate our approach, we introduce the example shown in Figure 1, which corresponds to a problem from the depots domain [17]. This problem consists in transporting three crates from their initial locations to their destination by means of a truck, which is initially at Depot0. A crate must be held by the hoist at the corresponding location to load it in the truck or unload it from the truck. Both Hoist0 and Hoist1 can be used to lift/drop crates in Depot0. Each hoist can only lift one crate at a time, whereas the truck has unlimited capacity.

Several decomposition approaches can be adopted to solve this problem as stated in Section 2. For example, the problem can be decomposed into three subproblems, one for each top-level goal. In this case, each subproblem is solved separately and then the obtained plans are combined. Plans for achieving these top-level goals are shown in Figure 2. A close analysis of these plans reveals that some conflicts will be found during the plan combination process. First, the same resources (hoists at each location and the truck) are used to reach different goals. We also have to take into account that Crate0 must be dropped onto Pallet2 before stacking Crate2 onto it. If we do not take this aspect into account, the plan for the top-level goal (on Crate2 Crate0) will be incorrect (as shown in Figure 2) because Crate2 is not dropped at its final correct location, which should be Distributor0 instead of Depot0 (where Crate0 is initially located).

These conflicts could be partly avoided if a **chronological decomposition** of the problem is considered, taking into account the intermediate positions that each object must reach throughout the execution of the plan. In our example, it can be observed that both Crate0 and Crate1 must be inside Truck0 at some point. Then, Truck0 must visit each of the crate destinations and, finally, the crates must be placed at their destination pallet (which must be clear). Figure 3 shows the intermediate positions that the objects must reach to solve this problem (we do not know the intermediate positions for Crate2 since we only know that Crate2 must be stacked on Crate0 but we do not have any information about which distributor). Thus, not only is a decomposition by goals performed, but a transversal decomposition is also done, which takes into account intermediate situations that must be reached in order to achieve each goal.

Note that we are not calculating a plan for each subgoal. Instead, we are just decomposing a problem through the intermediate positions that the objects must reach. Obviously, this type of decomposition leads to a sequential resolution, as one position is reached after the other. For example, Crate0 cannot be dropped onto Pallet2 unless it has been previously lifted by Hoist2. However, since it is possible to establish a partial ordering between the intermediate positions of different objects, we can avoid the majority of conflicts that appear when combining subplans. For instance, we can state that Crate0 must already be on Pallet2 before Crate2 is stacked onto Crate0 (see arrow in bold in Figure 3).

We can force some intermediate situations to be reached before others not only to avoid conflicts but also to benefit from positive interactions. In our example, both Crate0 and Crate1 have to be loaded into the truck at Depot0 in order to be transported to their final destinations. We will avoid visiting Depot0 twice if we state that both crates must be in Truck0 before it moves from Depot0 (see square in bold in Figure 3).

An intermediate position that an object in the problem will eventually reach throughout the plan execution can be defined by a fact that must be true at some state in the plan execution. In the literature, these facts are commonly known as **landmarks**. The work of Hoffmann, Porteous and Sebastia [24] is one example. This work also defines some types of order relationships between these landmarks, which are used to build a **landmark graph**.

The decomposition technique that we describe in this paper (called STeLLa) first studies the structure of the problem and computes a set of

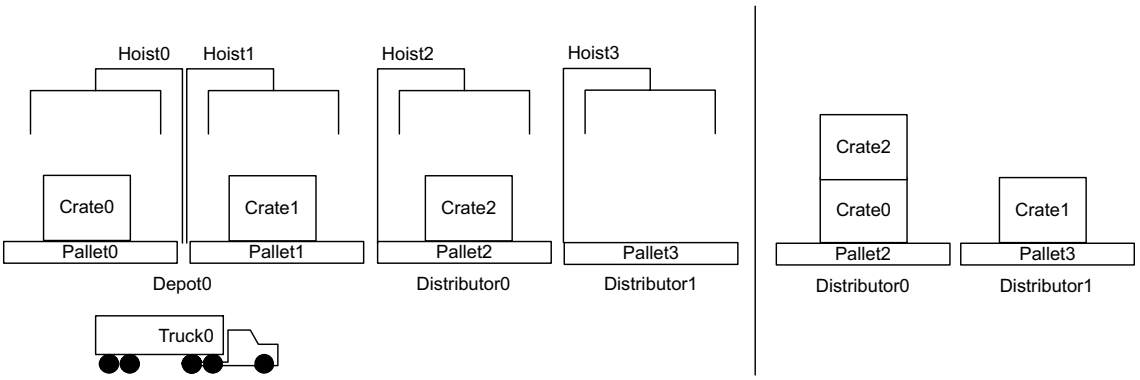

Fig. 1. Initial state (left) and goal state (right)

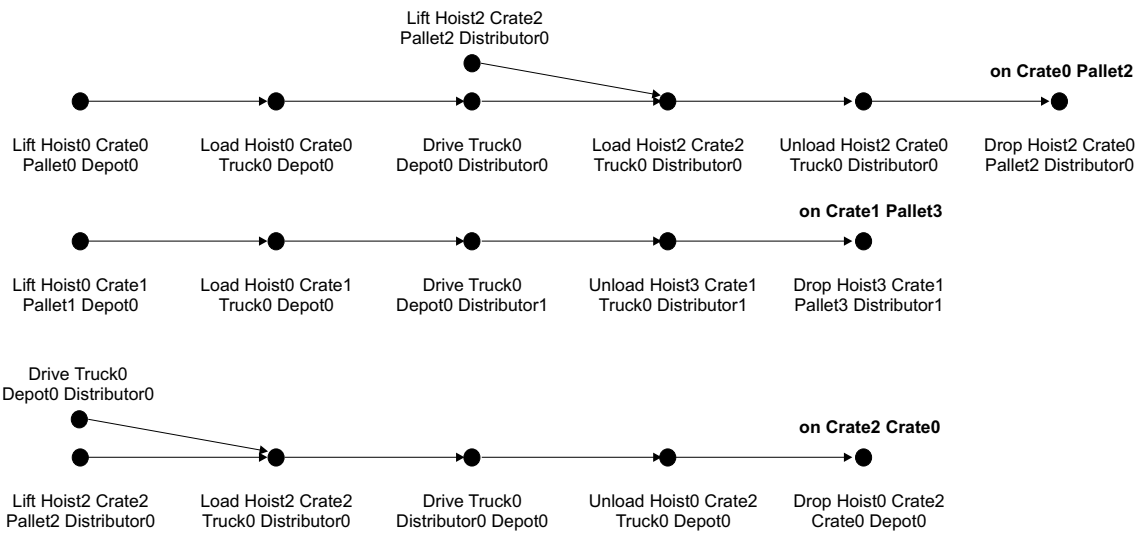

Fig. 2. Plans for each top-level goal

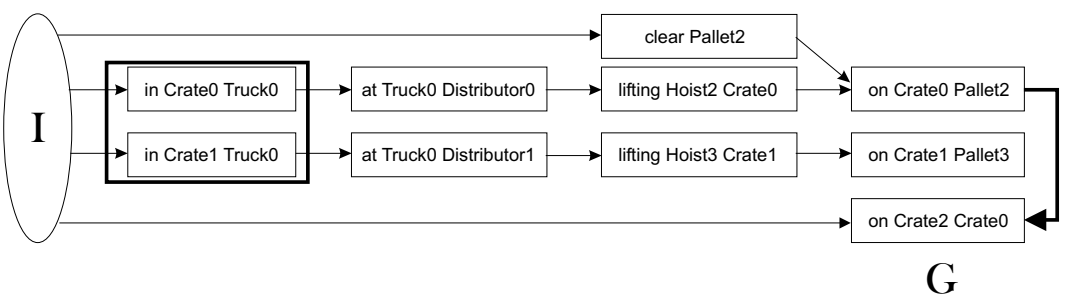

Fig. 3. Chronological decomposition of the planning problem

landmarks for each object. These landmarks are ordered to obtain a landmark graph, in a way similar to the one in [24]. The landmark graph is the basis of the process that is in charge of chronologically decomposing the original problem. This process analyses the landmark graph and finds addi-

tional ordering constraints between landmarks to avoid conflicts and to benefit from positive interactions. Thus, a collection of sets of landmarks is built (where each of these sets contains one or more landmarks) representing an intermediate situation in the planning problem. These sets are called *intermediate goals*.

Given a planning problem $P = \langle \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$, we obtain a set of subproblems of the form: $P_i = \langle \mathcal{A}, IS_{i-1}, IG_i \rangle$, where IG_i is an intermediate goal and IS_{i-1} is the state reached after solving the previous subproblem. In the first subproblem, $IS_0 = \mathcal{I}$; and in the last subproblem, $IG_n = \mathcal{G}$. These subproblems are solved by any planner to obtain a plan $\mathcal{P}_i = \langle a_{i1}, a_{i2}, \ldots, a_{ij} \rangle$. Solving each of these subproblems is easier than solving the original problem since the search space is vastly reduced. Once the solution for each subproblem is computed, the final solution plan for the original problem is obtained by simply concatenating the plans $\mathcal{P}_i$ above:

$$\mathcal{P} = \mathcal{P}_1 \circ \mathcal{P}_2 \circ \ldots \circ \mathcal{P}_{n-1} \circ \mathcal{P}_n$$

By using this technique, we avoid the conflicts that usually appear in a goal decomposition approach since these conflicts are handled in advance, when building the set of intermediate goals.

The following sections detail and formalize our decomposition technique. In Section 3.1, we introduce the concept of landmark and how landmarks can be ordered to build the landmarks graph. Section 3.2 explains how a planning problem is decomposed through the calculation of the intermediate goals. Section 3.3 shows the process of solving a planning problem decomposed with STeLLa; and Section 3.4 presents the experimental results comparing the resolution of a decomposed problem with the resolution of the original problem.

3.1. Detection and ordering of landmarks

The first step in our decomposition technique is the detection of landmarks for each object in the planning problem $\langle \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$. The work developed in [37] and [24] goes in this direction. These papers introduce the definition of landmark and some ordering relations between landmarks. STeLLa makes use of these ordering relations to build more informed and accurate intermediate goals. **Actually, Definitions 1 to 5 are borrowed from [24].** In this section, we summarize the main concepts about the theory of landmarks and orders[4]. We also introduce an improvement that was developed to find more precise orders.

Definition 1 *Given a planning problem* $\langle \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$, *a literal* l *is a* landmark *iff* l *is true at some point in all solution plans, i.e., if for all* $P = \langle a_1, \ldots, a_n \rangle \in \mathcal{A}^*, \mathcal{G} \subseteq Result(\mathcal{I}, P) : l \in Result(\mathcal{I}, \langle a_1, \ldots, a_i \rangle)$ *for some* $0 \leq i \leq n$.

As mentioned above, it is possible to establish some orders between landmarks, thereby indicating which landmarks should be achieved before others in the plan. Consequently, the result of this process is a graph where nodes represent landmarks and edges represent orders between them.

Definition 2 *Given a planning problem* $\langle \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$ *and two literals* l *and* l', *there is a* **necessary** *order between* l *and* l', *denoted as* $l \leq_n l'$, *if* l *is a prerequisite in the immediately preceding state for achieving* l':

$$\begin{aligned} & l' \notin \mathcal{I} \wedge (\forall \mathcal{P} = \langle a_1, \ldots, a_n \rangle \in \mathcal{A}^* : \\ & \quad l' \in Result(\mathcal{I}, \langle a_1, \ldots, a_i \rangle) \ \wedge \\ & \quad l' \notin Result(\mathcal{I}, \langle a_1, \ldots, a_j \rangle), 1 \leq j < i \leq n) \\ & \rightarrow (l \in Result(\mathcal{I}, \langle a_1, \ldots, a_{i-1} \rangle)) \end{aligned}$$

A necessary ordering established between l and l' indicates that l is a prerequisite for l' only the first time l' is achieved in the plan.

We can also establish a **reasonable** order between two literals, l and l', which states that it is reasonable to achieve l first, because, otherwise, l' would be achieved twice in the plan. The following definitions formalize this notion of reasonable order.

Definition 3 *Given a planning problem* $(\mathcal{A}, \mathcal{I}, \mathcal{G})$ *and two literals* l *and* l'. l interferes *with* l' *if one of the following conditions holds:*

1. l *and* l' *are inconsistent*[5];

[4] The complete theory about landmarks can be found in [24].

[5] Two literals are said to be inconsistent when they cannot simultaneously coexist in the same correct planning state. We approximate inconsistency by using the function *inconsistent* provided by the TIM API[15]. **If the function** *inconsistent*(l, l') **returns** True, l **and** l' **are inconsistent, but if it returns** False **it is not known if** l **and** l' **are inconsistent (only that the function failed to prove them to be.**

2. *there is a literal* $x \in \bigcap_{a \in \mathcal{A}, l \in add(a)} add(a)$ *such that* x *is inconsistent with* l'*;*
3. $l' \in \bigcap_{a \in \mathcal{A}, l \in add(a)} del(a)$*;*
4. *or there is a landmark* x *that is inconsistent with* l' *such that* $x \leq_n l$*.*

Definition 4 *Given a planning problem* $(\mathcal{A}, \mathcal{I}, \mathcal{G})$*, and two landmarks* l *and* l'*. If* l *interferes with* l'*, and there is a set of landmarks* $l_1, \ldots, l_n$ *such that* $l_i \leq_n l_{i+1}, 1 \leq i < n$*, with* $l = l_1$ *and either*

1. $l', l_n \in \mathcal{G}$*, or*
2. *there is a landmark* l_{n+1} *such that* $l' \leq_n l_{n+1} \wedge l_n \leq_n l_{n+1}$*,*

then there is a reasonable ordering between l *and* l'*, denoted as* $l \leq_r l'$*.*

In a similar way as necessary orders are used to define reasonable orders, both necessary and reasonable orders are used to define orders of a new type: the **obedient** orders.

Definition 5 *Given a planning problem* $(\mathcal{A}, \mathcal{I}, \mathcal{G})$*, a set of necessary and reasonable orders* O*, and two landmarks* l *and* l'*, if* l *interferes with* l'*, and there is a set of landmarks* $l_1, \ldots, l_n$ *such that* $(l_i, l_{i+1}) \in O, 1 \leq i < n$*, with* $l = l_1$ *and either*

1. $l', l_n \in \mathcal{G}$*, or*
2. *there is a landmark* l_{n+1} *such that* $l' \leq_n l_{n+1} \wedge (l_n, l_{n+1}) \in O$ *and* $\exists i : 1 \leq i \leq n$ *such that there is no action* $a \in \mathcal{A}$ *with* $l_i \in add(a) \wedge l_{i+1} \in add(a)$

then there is an obedient ordering between l *and* l'*, denoted as* $l \leq_o l'$*.*

Once the landmarks and the necessary orders between them have been found, definitions 4 and 5 are applied to find the reasonable and obedient orders between these landmarks. The combination of all these types of orders defines a landmark graph.

Definition 6 *A* **landmark graph (LG)** *is a graph* (L, E) *where* L *is the set of the extracted landmarks and* E *represents the necessary, reasonable and obedient orders among landmarks together with the type of order:*

$$E = \begin{cases} (l_i, l_j, n) & : \quad l_i \leq_n l_j \\ (l_i, l_j, r) & : \quad l_i \leq_r l_j \\ (l_i, l_j, o) & : \quad l_i \leq_o l_j \end{cases}$$

We represent the existence of an edge between two landmarks l *and* l' *in the LG as* $l \leq l'$*.*

Figure 4 shows the LG for the example in Figure 1. We can observe that there are no landmarks indicating which hoist must be used to lift Crate0 and Crate1 from Pallet0 and Pallet1, respectively, because both Hoist0 and Hoist1 can be used. Consequently, the first intermediate position reached by these crates indicates that they must be in the truck: (in Crate0 Truck0), (in Crate1 Truck0). In addition, when Crate0 has to be unloaded from Truck0 at Distributor0, only Hoist2 can be used and, therefore, literal (lifting Hoist2 Crate0) appears as a landmark. A similar situation occurs with Crate1.

The only landmarks that are related to Crate2 are those from the initial state and the top-level goals. This occurs because we can achieve (on Crate2 Crate0) at any depot or distributor and, therefore, any hoist can be used to drop Crate2 onto Crate0. This lack of information in the goal description makes the process for landmark extraction be unable to find more landmarks that are related to Crate2 such as (lifting Hoist2 Crate2), which is a literal that must be clearly achieved in order to clear Pallet2. This is an indication of the incompleteness of the process for landmark extraction. However, in this particular case, we can indirectly discover this landmark since (clear Pallet2) is a landmark which can only be reached if Crate2 is lifted.

Let us give an example of reasonable and obedient orders. It is obvious that l =(on Crate0 Pallet2) must be achieved before l' =(on Crate2 Crate0). This reasonable order is found as follows: l interferes with l' because the only action that achieves l has an add effect ((clear Crate0)), which is inconsistent with l' (condition 2 of Definition 3), and both l and l' belong to $\mathcal{G}$ (condition 1 of Definition 4). We can establish an obedient order between l =(lifting Hoist2 Crate0) and l' =(clear Crate0) in the following way. l interferes with l' because they are inconsistent (condition 1 of Definition 3). As $l \leq_n$ (on Crate0 Pallet2) $\leq_r$ (on Crate2 Crate0) and $l' \leq_n$ (on Crate2 Crate0), then we can establish the obedient order $l \leq_o l'$.

It is important to note that the information contained in the LG might be incomplete because the algorithms for finding landmarks and orders do not guarantee finding either all the possible landmarks nor all the necessary, reasonable and obedient orders. In addition, the introduction of some reasonable and obedient orders might produce cycles in the LG. Unlike necessary orders, reasonable and

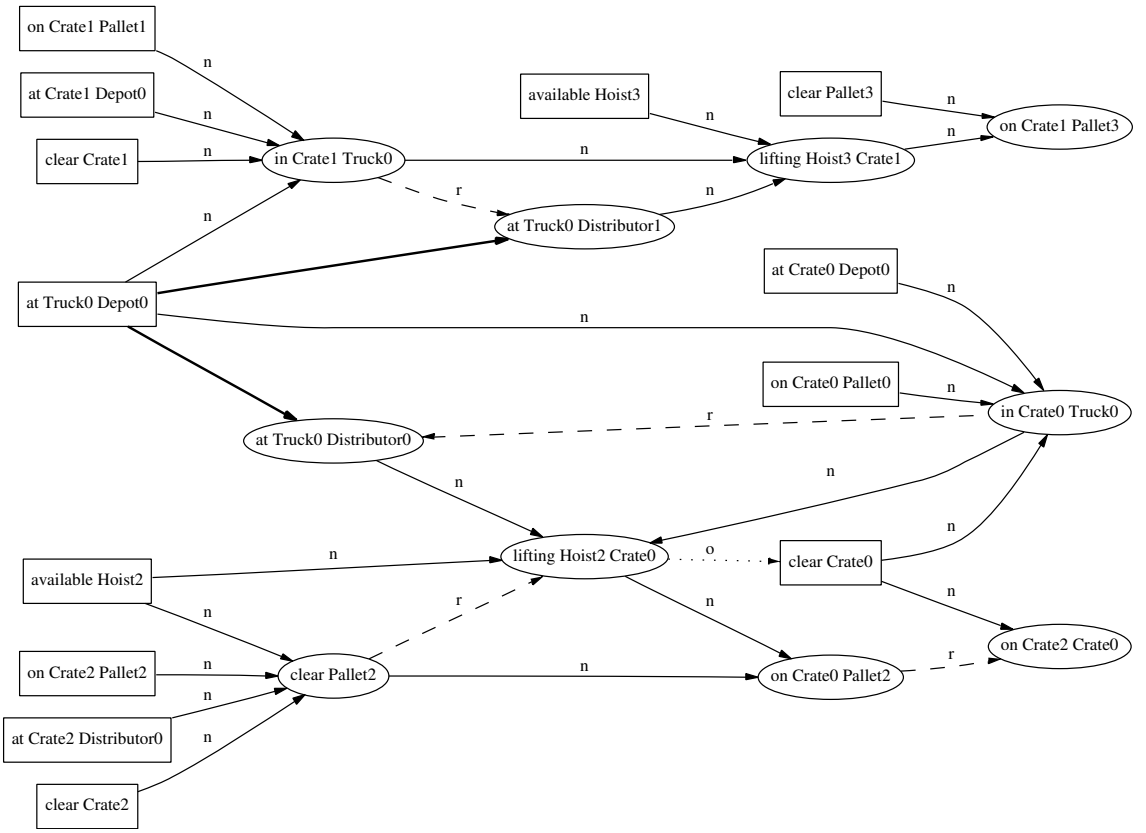

Fig. 4. The landmark graph. Literals from the initial state are represented with boxes; necessary orders are indicated by solid lines; reasonable orders are indicated by dashed lines; and obedient orders are indicated by dotted lines.

obedient orders work as *recommendations* about which literals would better be achieved first and they do not impose a restriction on how literals should be ordered. This might give rise to a situation where a literal l is ordered before another literal l', even when l is reachable from l'. For instance, Figure 4 contains a cycle formed by literals {(clear Crate0), (in Crate0 Truck0), (at Truck0 Distributor0), (lifting Hoist2 Crate0)}, as a consequence of the existence of an obedient order between (lifting Hoist2 Crate0) and (clear Crate0) and also a path from (clear Crate0) to (lifting Hoist2 Crate0). A cycle indicates that a landmark will appear more than once in the plan[6]. In this example, the cycle represents that the literal (clear Crate0) will appear twice: the first time in the initial state, and the second time when Crate0 is dropped onto Pallet2.

[6]The appearance of cycles in the LG and their meaning will be discussed in the following section.

When the landmark extraction process described in [24] is applied, the literals (at Truck0 Distributor0) and (at Truck0 Distributor1) have (at Truck0 Depot0) (the initial location of the truck) as a predecessor node (lines in bold in Figure 4 represent this relationship). This would imply that to reach (at Truck0 Distributor0) and (at Truck0 Distributor1), the origin of Truck0 must be Depot0, which is only true the first time the truck moves. Therefore, the process described in [24] obtains *incorrect* necessary orders. In order to overcome this drawback, we have implemented a **postprocessing** to remove these orders, **which is executed right after having computed the necessary orders (that is, before calculating the reasonable and obedient orders)**. It basically consists in deleting those necessary orders (l, l') such that l' can be reached by using an action that does not require l as a precondition. This is why the literals (at Truck0 Distributor0) and (at Truck0 Distributor1) do not have any predecessor landmark in the graph (lines in bold no longer form part of the LG).

3.2. Intermediate goals

This section describes how a problem is partitioned when following our decomposition technique. STeLLa performs a chronological decomposition by taking into account the following two aspects:

1. the intermediate positions that each object in the domain must reach throughout the plan execution
2. the negative interactions that may appear between the actions that solve each top-level goal.

This information is captured by the LG. Intermediate positions of objects are represented by landmarks, while negative interactions are denoted by reasonable and obedient orders. For example, the reasonable order between (in Crate0 Truck0) and (at Truck0 Distributor0) states that the action that achieves the second landmark deletes a literal required by the action that achieves the first landmark. All this information is reflected in the LG and is used as a basis in our decomposition technique. Once we have the LG for a given problem, STeLLa analyses its structure to compute a set of sequential intermediate goals.

Definition 7 *An* **intermediate goal (IG)** *is a set of landmarks from the corresponding $LG(L, E)$ that fulfil the following properties:*

- Consistency Property. *All the literals in an IG must be consistent with each other:*

$$\forall l, l' \in IG : \neg inconsistent(l, l')$$

- Ordering Property. *A literal l belongs to an IG if and only if all its predecessor nodes in the $LG(L, E)$ have been visited*[7] *before l:*
$\forall l \in IG_i : \forall l' \in L : l' \leq l \rightarrow l' \in IG_j \wedge j < i$

The IGs are built by going through the LG and by grouping together those landmarks that fulfil the *Consistency* and *Ordering* properties. Once the IGs have been computed, we build a set of subproblems of the form $P_i = \langle \mathcal{A}, IS_{i-1}, IG_i \rangle$, where IG_i is an IG, and IS_{i-1} is the state reached after solving the previous subproblem. The IG is the goal set of a subproblem; the *Consistency* Property must be satisfied for all the IGs to ensure that each of them will be a correct goal set. Also, the *Ordering* Property forces building the IGs according to the orders in the LG. These orders reflect two types of relationships between two landmarks l and l': a necessary order represents that to achieve l', l must be true in the previous state, while reasonable and obedient orders denote a negative interaction between the actions that achieve both landmarks. Therefore, by obeying the orders in the LG, we will be able to build more accurate IGs.

All the literals in the LG will eventually be included in an IG and, consequently, they will be achieved at some point of the plan. For this reason, the LG must only contain literals labeled as landmarks and not other literals since this may affect the completeness of the process and also the quality of the solution. These two factors can also be affected by the order in which landmarks are included in the IGs. Let us recall that the LG is incomplete because the process of landmark extraction does not obtain all the possible orders between landmarks. This implies that, when two landmarks l and l' are not ordered, we can generate three different decompositions: in the first one, l is included in IG_i and l' in IG_j, such that $i < j$; in the second solution, l belongs to IG_j and l' to IG_i; and in the last one l and l' belong to the same IG_i. In the first case, l would be achieved before l' in the plan and in the second case, l' would be achieved before l. In the third case, it is the planner that takes the decision about which literal to achieve first. This may affect completeness (if we deal with non-reversible domains [22]) as well as the solution quality. For this reason, we introduce some helpful additional constraints to build more accurate IGs and obtain a more refined problem decomposition.

These additional constraints are computed at the same time the IGs are being built. The objective is to establish some priority criteria to promote some literals over others when inconsistencies among landmarks are found. These additional constraints capture and deal with the negative interferences that have not been detected during the search for orders among landmarks. We call these constraints *active interferences*. **More particularly, the aim of active interferences is to make the necessary modifications to the IGs so as to facilitate the fulfilment of the Consistency and Ordering properties. We distinguish four cases (depicted in Figure 5). The first case provides support for guar-**

[7] A literal is said to have been visited if it has been included in a previous IG.

anteeing the Ordering property; the other three cases provide support for guaranteeing both properties. Let IG_{i-1}, IG_i and IG_{i+1} be three consecutive IGs, and let l and l' be two landmarks that belong to LG(L,E).

a) **If l belongs to IG_{i-1}, l will be propagated to IG_i (that is, it will be included in IG_i too) until a successor literal l' in the $LG(L,E)$ (through a necessary order) is visited:**
$\forall l \in IG_{i-1} : (\forall l' \in L : l \leq_n l' \wedge l' \notin IG_i) \Rightarrow IG_i = IG_i \cup \{l\}$
If none of the successor literals l' of l linked with a necessary order has been visited, l will be propagated (it will be included in the next IG) and so on until a successor literal is visited. The reason for this is to prevent l from being deleted before the successor literal l' is included in an IG because, if this were the case, then l would have to be achieved again before satisfying l'.

b) **If two inconsistent landmarks l and l' belong to IG_i and l' is a propagated literal, l is delayed to IG_{i+1} (that is, it is only included in IG_{i+1}):**
$\exists l, l' \in IG_i : inconsistent(l, l') \wedge l' \in IG_{i-1} \Rightarrow \begin{cases} IG_i = IG_i - \{l\} \\ IG_{i+1} = IG_{i+1} \cup \{l\} \end{cases}$
If a propagated literal l' is inconsistent with another literal l, then l' is given more priority for the reasons explained in the case above, and l is the landmark to be delayed.

c) **If two inconsistent landmarks l and l' belong to IG_i, and there is a literal l_p in the previous IG so that $l_p \leq_n l'$, then l is delayed to IG_{i+1}:**
$\exists l, l' \in IG_i : inconsistent(l, l') \wedge \exists l_p : (l_p \in IG_{i-1} \wedge l_p \leq_n l') \Rightarrow \begin{cases} IG_i = IG_i - \{l\} \\ IG_{i+1} = IG_{i+1} \cup \{l\} \end{cases}$
If a literal l' is inconsistent with another literal l, and l' is linked through a necessary ordering with a literal in the previous IG, then l' is given more priority, and l is the landmark to be delayed.

d) **If two landmarks l and l' belong to IG_i, and there is a landmark l'' so that $l'' \leq_n l'$ and l'' is inconsistent with l, then l is delayed because the plan should achieve l' first in order not to delete l'' with l.**
$\exists l, l' \in IG_i \wedge \exists l'' : l'' \leq_n l' \wedge inconsistent(l, l'') \Rightarrow \begin{cases} IG_i = IG_i - \{l\} \\ IG_{i+1} = IG_{i+1} \cup \{l\} \end{cases}$
Again more priority is given to those necessary orderings that are already established, so that in case of conflict, the landmark to be delayed is always the literal that does not hold any necessary ordering with the literals in the previous IG.

The IGs are computed by going through the LG and by grouping together those landmarks according to the active interferences explained above. Taking into account that $IG_0 = \mathcal{I}$, the process of building IG_i works as follows:

1. First, an approximation to IG_i is computed with all the landmarks l that have a predecessor literal l' in IG_{i-1}:
$$IG_i = \{l \in LG / \exists l' \in IG_{i-1} : l' \leq l\}$$
2. Second, we refine this first approximation in three stages:
   (a) Delay landmarks l that have a predecessor literal l' in the LG such that l' has not been visited (*Ordering* Property).
   (b) Propagate the corresponding literals from IG_{i-1} to IG_i (the first case of active interferences).
   (c) Check the remaining active interferences between the literals in IG_i.

A special case is applied to literals from $\mathcal{G}$. Once a literal g from $\mathcal{G}$ has been included in an IG, g will be propagated unless an inconsistent landmark is added in the same IG. In this case, g will be delayed to the last IG.

3.2.1. Cycles in the LG

One problem that we come across during the construction of the IGs is due to the existence of cycles in the LG. Cycles lead into a deadlock because none of the landmarks involved in the cycle satisfy the *Ordering* property. For example, in Figure 4, we find the following cycle: C={(in Crate0 Truck0) $\leq_r$ (at Truck0 Distributor0) $\leq_n$ (lifting Hoist2 Crate0) $\leq_o$ (clear Crate0) $\leq_n$ (in Crate0 Truck0)}, where all the literals have a unvisited predecessor.

Definition 8 *A* **cycle** *in a $LG(L,E)$ is defined as a path whose initial and final node coincide:*
$l_1, \ldots, l_n, l_{n+1} \in L / \forall 1 \leq i \leq n : (l_i, l_{i+1}) \in E \wedge l_1 = l_{n+1}$

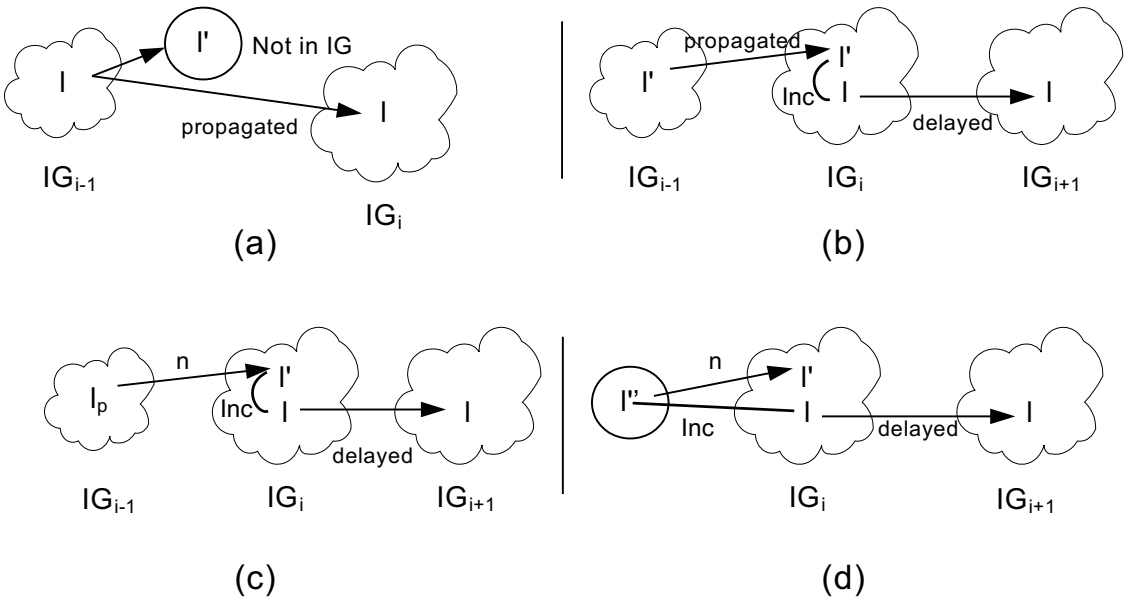

Fig. 5. Cases of active interference

A literal l that is involved in a cycle will always be delayed by step 2a of the refinement process in the IG construction; thus l will never be included in an IG. A cycle provides very relevant information as it points out that a least one of the literals involved in the cycle must be achieved at least twice. In the example above, (clear Crate0), a landmark from the initial situation, is ordered **after** another landmark. This clearly indicates that (clear Crate0) will appear at least twice in the problem: once as a literal from the initial situation and later, when Crate0 is dropped onto Pallet2. Therefore, the behaviour of our algorithm must be modified accordingly in order to exploit the information provided by the cycles and, at the same time, to avoid the deadlocks that cycles may cause. This modification aims at selecting *the cycle start point* from the landmarks in the cycle. The cycle start point is the literal that will be visited both first and last (at least twice). Once the cycle start point, l, is identified, l is included in the IG, and the remaining literals in the cycle will be visited in the corresponding order. Finally, l will be visited again as the last literal in the cycle.

The cycle start point must be carefully selected so as to minimize the number of landmarks in the cycle that are visited twice unnecessarily. This process is performed in two steps. The first step builds the set of possible start points, which is the set of literals whose incoming edges from the landmarks in the cycle are not necessary orders. Note that a necessary order forces the predecessor node to be included in the previous IG, so the successor node can never be the cycle start point.

Definition 9 *Given a cycle C of an LG, the* **set of possible start points** *-$SP_0(C)$- is defined as follows:*

$$SP_0(C) = \{l \in C / \neg \exists l' \in C : l' \leq_n l\}$$

The second step refines the set $SP_0(C)$ and builds a set of start points ($SP(C)$) rather than selecting a single start point from $SP_0(c)$, which might likely lead to an incorrect decision. By previously computing a set of start points $SP(C)$, the mechanism for making the final decision becomes more trivial because the first literal reached in this set will be considered as the start point of the corresponding cycle. In other words, when a literal l in $SP(C)$ is selected to be visited, l is included in an IG even though the Ordering Property is not satisfied. The set $SP(C)$ is composed of those landmarks that have a higher priority to be reached twice in a plan, particularly those from the initial situation or from the goal set. In first place, the process for building $SP(C)$ checks the landmarks in the cycle that belong to $\mathcal{I}$ ($SP(C) = \{l \in SP_0(C) / l \in \mathcal{I}\}$) as these land-

marks are very likely to be achieved twice; once in the initial state and later during the execution of the plan. This is the case presented in Figure 4. If no literal in the cycle belongs to $\mathcal{I}$, the algorithm checks in second place the literals that belong to the goal set ($SP(C) = \{l \in SP_0(C)/l \in \mathcal{G}\}$). A literal l from the goal set appears in $SP_0(C)$ when there exists another literal l' in the cycle such that $l' <_r l$ or $l' <_o l$ (Definition 9), which means that achieving l' after l would delete the latter. Since l belongs to $\mathcal{G}$, it seems reasonable to visit l in the last place in the cycle and, therefore, it should be visited in the first place, too. Finally, if no literal belongs to $\mathcal{I}$ or $\mathcal{G}$, then $SP(C) = SP_0(C)$.

The above selection criteria identify those literals which are the most likely to be achieved more than once in the plan. A landmark from the initial situation is first chosen as the cycle start point; otherwise, a landmark from the goal set is chosen as the cycle final point, and it will consequently also mark the start point of the cycle. An incorrect selection of the cycle start point would inevitably lead to a degradation in the solution quality as it would require more landmarks to be visited more than once.

Once we have selected the cycle start point for each cycle in the LG, the process for computing the IGs is slightly modified. The application of this modification may share a certain similarity with the resolution of the Feedback Vertex Problem [18] in the sense that when a literal l is selected as the cycle start point all the incoming edges from other literals in the cycle are temporally ignored. This way, if l satisfies the remaining properties and constraints, l will form part of an IG (even though it has non-visited predecessor literals). Once l is included in an IG, the ignored edges are reconsidered and become part of the LG again, which will eventually lead literal l to be visited for the second time.

We have experimentally confirmed that the proposed criteria produce better quality plans than random selection, for the domains used in our experiments (see Section 3.4). Several tests were performed on randomly chosen landmarks involved in the cycle. The results show that better quality solutions were always obtained when the landmark that was selected as the start cycle point was a literal from the initial situation, or when it was a literal from the top-level goal.

3.3. Resolution of decomposed problems

In the previous section, the main features of STeLLa were described in detail. The final goal of this decomposition technique is to obtain a plan that solves the original problem. Once the intermediate goals have been computed, a set of sequential subproblems is built where each IG represents the goal set of each subproblem. Figure 6 shows the resolution scheme. The initial state of the first subproblem is the initial state of the original problem $\mathcal{I}$, and the goal set is the first IG, IG_1. Any STRIPS planner that solves this first subproblem returns a plan $\mathcal{P}_1$. The following initial state IS_1 (*intermediate state*) is obtained as the result of applying $\mathcal{P}_1$ to I. The second subproblem is composed of this new initial state IS_1 and the next IG, IG_2, as the goal set. Again, this new subproblem is solved by a planner to obtain $\mathcal{P}_2$, which is used together with IS_1 to compute IS_2. This process continues in the same way until the last IG, IG_n, is used as the goal set of a subproblem. Once the plan $\mathcal{P}_n$ is computed, the last subproblem is built with IS_n and the goal set of the original problem $\mathcal{G}$. The solution plan $\mathcal{P}$ for the original problem is computed by concatenating plans $\mathcal{P}_i$.

This solution plan $\mathcal{P}$ is **correct** provided that the solution to each subproblem P_i is produced by a correct planner (we assume this is the case). Moreover:

- The initial state of each $\mathcal{P}_i$ is a correct state as this is the result of applying the last computed subplan to the previous subproblem.
- The goal set of the last subproblem is the goal set of the original problem $\mathcal{G}$. Therefore, if $\mathcal{G}$ is not achieved through the resolution of the different intermediate subproblems, $\mathcal{G}$ will be achieved when solving the last subproblem.

For all these reasons, we can state that the combination of our decomposition technique with a correct planner will produce correct solutions.

However, our decomposition technique cannot be guaranteed to be complete for two reasons:

1. *The construction of non-reachable IGs due to multi-mutex relations.* In contrast with binary mutex relations (that is, when two literals are inconsistent), multi-mutex relations [15] indicate that larger groups of literals are collectively incompatible. The process of

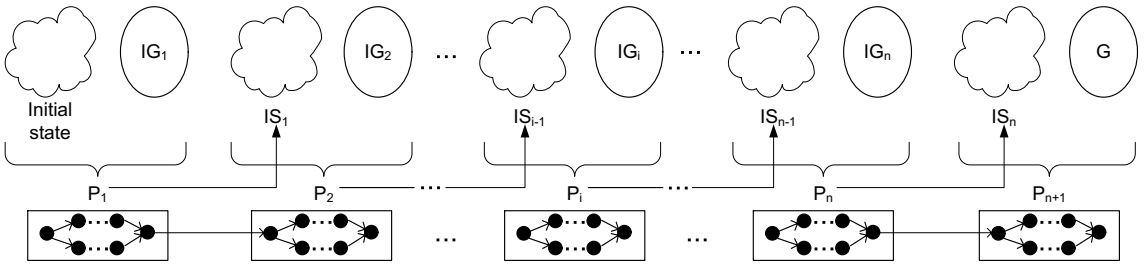

Fig. 6. Resolution scheme

building the IGs only checks binary mutex relations, so IGs with multi-mutex relations may be built. However, in our experiments, we have never found such a case.

2. *The dead-ends originated in non-invertible domains.* A dead-end is a reachable state from which the top-level goals can no longer be reached [29]. Obviously, invertible domains are dead-end free because from any state that is reached after applying action a, it is always possible to apply its counterpart action a', which undoes a's effects [22]. When we deal with non-invertible domains, we may generate an IG from which the goals can no longer be reached. This situation arises for several reasons; however, the most common reason is when a literal l is included in an IG that is previous to the correct one. In this case, l is achieved before other literals that cannot be reached from a state where l is true. This occurs because the LG does not contain essential information about some orders between landmarks, which may prevent the process of building the IGs to include a literal in the right IG.

In spite of this incompleteness, the experiments shown in the following section will demonstrate that STeLLa is able to solve more problems than other planners or other decomposition techniques.

The lack of information in the LG also affects the solution quality (both in invertible and non-invertible domains) because, again, some literals may be included in incorrect IGs. This is partially overcome thanks to the active interferences computed during IG construction. By building a conjunctive set of goals (intermediate goals), our decomposition technique takes advantage of positive interactions between landmarks, which also improves the quality of the solutions. If there is a positive interaction between two landmarks, we can exploit this advantage if the two landmarks are in the same IG. Otherwise, this advantage may be lost.

3.4. Experiments

In this section, we present the experiments that were performed to observe the behaviour of STeLLa in solving the IPC2 and IPC3[8] benchmark suites[9]. The tests were conducted on a Pentium 4, 2.0 GHz with Linux RedHat 2.7 and a main memory of 1 Gbyte. We show two groups of experiments:

- In Section 3.4.1, we compare the performance of three different planners (FF[21], LPG[20] and VHPOP[45]) when solving the original problem and the partioned problem by STeLLa. We have selected these planners because they are based on different planning approaches and have been recognized in the planning competitions as outstanding planners. FF is a heuristic planner, LPG is based on local search, and VHPOP is a partial-order planner.
- In Section 3.4.2, we compare STeLLa (using FF as a base planner) with another two decomposition techniques, FF-L and SGPlan, whose main features are explained in Section 2.

First, it is important to analyse the *level of decomposition* of the different domains, that is the number of IGs that can be obtained when using our decomposition technique. The number of IGs depends on how many landmarks and orders are obtained for a certain problem, so the level of decomposition cannot be set a priori. The number

[8] IPC stands for International Planning Competition.

[9] A complete description of these domains can be found in the Web sites of IPC2[2] and IPC3[17]

of extracted landmarks depends on whether the various alternatives (sequences of actions) for obtaining a particular literal share common requirements (action preconditions); these common requirements will be part of the landmarks set. Also, the number of orders established between landmarks depends on the number of interactions between them.

Figure 7 (left) shows the number of IGs generated for the blocksworld, elevator, freecell and logistics domains. Figure 7 (right) shows the number of IGs generated for the depots, driverlog, satellite and zeno domains. The domain in which the most IGs are obtained is the blocksworld domain. This is because of the large number of landmarks and the highly- interactive top-level goals. In this domain, many orders are generated between landmarks because top-level goals have to be reached in a specific order with only one robot arm. A similar analysis can be made for the elevator domain (Figure 7 left) and the depots domain (Figure 7 right). In the freecell domain, the obtained landmarks basically come from the initial and goal state, so it is almost impossible to identify intermediate situations. The IGs for this domain are due to the orders found between the top-level goals. This is actually not very helpful for solving the partitioned problem, as we will see in the following section. In summary, the more interactions between goals and subgoals, the greater the number of IGs that can be obtained. This is especially the case for domains that have only one non-shared available resource.

In the logistics domain, there are two types of problems: (1) those where only one airplane is available (until instance 22) and (2) those where there are several airplanes to fly between cities. In the latter case, fewer landmarks can be found because there are more alternatives to solve the problem, which also determines that there will be fewer interactions between landmarks. This fact is particularly noticeable in domains such as driverlog (Figure 7 right), where several alternatives (several trucks and several intermediate cities) can be used for the same purpose; consequently, only one IG (which coincides with the goal set) is obtained. A similar situation occurs in the satellite and zeno domains. In domains that cannot be decomposed, solving the partitioned problem is equivalent to solving the original problem.

Table 3 shows a breakdown of the average time required to decompose the problem. It can be observed that, in general, decomposing a planning problem is not very time-consuming. However, in the blocksworld domain, where the obtained LGs are very informative, the processes for computing the orders between landmarks and for building the IGs are more costly. This makes up for the fact that we obtain simpler problems as the results will show. For some domains (freecell and zeno), where the obtained IGs are not helpful, the IG construction takes considerable time due to the fact that in these domains the LG is not informative in terms of orders; therefore, many active interferences need to be computed.

3.4.1. FF, LPG *and* VHPOP

In this section, the performance of three planners (FF, LPG, and VHPOP) is compared when they are used to solve the original problem and the set of subproblems generated by STeLLa. The aim of this comparison is (1) to verify whether the decomposition facilitates the resolution of the problem (by comparing the number of problems solved by the two configurations), (2) to evaluate how the solution quality is affected when STeLLa is used and (3) to analyse in which domains and with which type of planners our decomposition technique is more efficient.

Table 4 shows the number of problems solved by each planner in each domain. The second line in each row correspond to our decomposition technique using the indicated planner as a base planner. Figures 8 to 11 show the performance in solving instances from IPC2 and IPC3. The X axis in these charts represents the results (the makespan and the CPU time) which were obtained when STeLLa-planner solved the partitioned problem. The Y axis represents the results of these planners when solving the original problem. The symbols above the line indicate that the results obtained by STeLLa-planner were better than those obtained with the original planners themselves. To complete the experiments within a reasonable amount of time, we restricted the time consumption to 100 seconds for FF and LPG, and we restricted the memory consumption to 512 Mb for VHPOP[10]. The results of LPG were obtained as the average of 5 runs.

The most outstanding result is that the three planners solve more problems when they are ex-

[10]This is due to the fact that VHPOP exhausted memory before the CPU time limit.

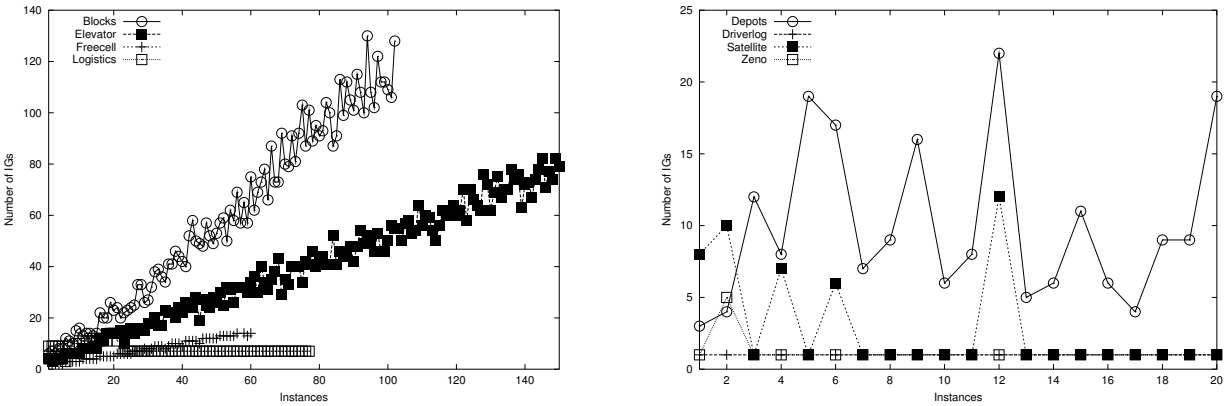

Fig. 7. Number of IGs generated for each instance in domains from IPC2 (left) and IPC3 (right) planning competitions.

	Blocksworld	Elevator	Freecell	Logistics	Depots	Driverlog	Satellite	Zeno
Landmarks	0.17	0.61	0.85	0.28	0.08	0.03	0.03	0.05
Orders	15.9	1.31	0	0.31	0.08	0	0	0
IGs	3.3	1.17	4.84	0.74	0.24	0.07	0.26	4.24

Table 3
Breakdown of decomposition time (secs.)

	Blocks	Elevator	Freecell	Logistics	Depots	Driverlog	Satellite	Zeno
Total	102	150	60	77	20	20	20	20
FF	77	150	50	77	17	15	20	20
STeLLa-FF	102	150	48	77	20	15	18	20
LPG	48	150	10	77	20	20	20	19
STeLLa-LPG	86	150	18	77	20	19	20	19
VHPOP	0	52	0	71	2	4	11	9
STeLLa-VHPOP	84	150	10	77	18	6	13	10

Table 4
Number of problems solved by FF, LPG, and VHPOP for the domains of the IPC2 and IPC3 planning competitions.

ecuted with our decomposition technique (Table 4). This was specially notable in VHPOP[11], which was able to solve 368 problems (out of 469) using STeLLa and only 149 when it was executed with the original non-partitioned problem. This implies that our decomposition technique decomposes the original problem into simpler problems, which can then be handled more easily by these planners.

In addition, though solving a partitioned problem usually implies a lack of quality in the solutions (since the problem is not being considered as a whole), our results show that the decomposition method used by STeLLa allowed us to preserve the quality of the solutions and even obtain shorter plans in many cases. In particular, FF and STeLLa-FF computed solutions of almost identical quality for most of the domains. Figure 8, Figure 9 and Figure 11 show that the results obtained by FF were always on or very close to the line. The results also show that STeLLa-FF outperformed FF in the depots domain in terms of solution quality (Figure 10). Similar conclusions can be drawn when comparing VHPOP and STeLLa-VHPOP. It is especially relevant to note that STeLLa obtained better quality solutions for the elevator and depots domains (Figure 8 and Figure 10).

With regard to the execution time of partitioned problems, it must be taken into account that the time spent parsing all of the subproblems

[11] There are very few comparisons of the STeLLa execution time with respect to VHPOP because it solved very few problems.

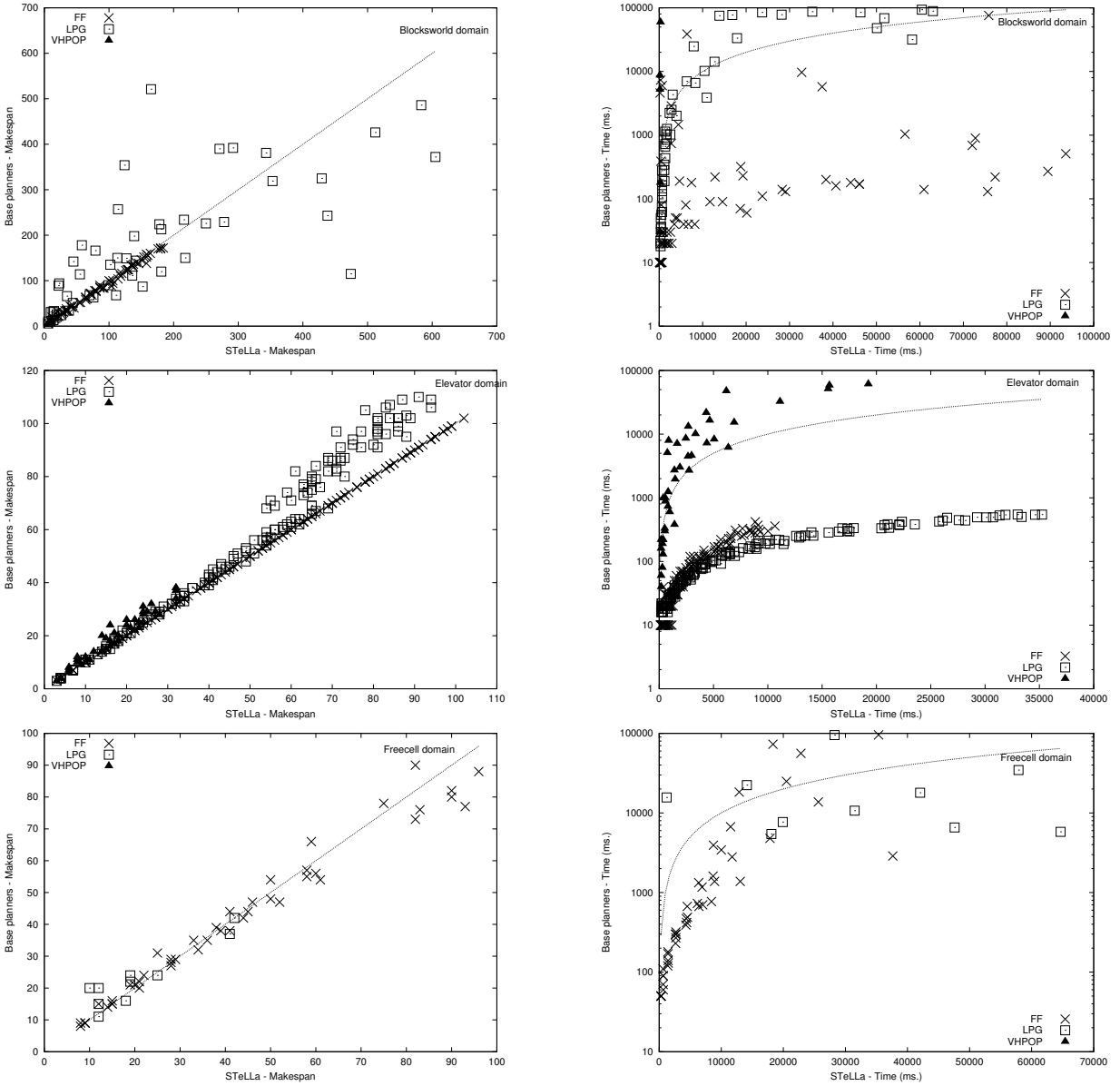

Fig. 8. Comparison between STeLLa and FF, LPG, and VHPOP in the blocksworld, elevator and freecell domains.

must be added to the decomposition time. That is, after the problem has been partitioned, the base planner has to parse each subproblem to solve it[12]. This factor made STeLLa much slower than FF and LPG for most of the domains (Figure 8, Figure 9 and Figure 11) although STeLLa ran faster for some problems in the depots domain (Figure 10). In other cases, just the decomposition of the problem took longer than solving the original one. This occured, for example, in the blocksworld domain with FF (Figure 8). Table 3 shows that decomposition was costly in this domain; however, in this case, the increase in the number of solved problems made up for the overhead of problem decomposition. In contrast, LPG and VHPOP improved their performance when using STeLLa in most problems of the blocksworld domain, in spite

[12]This overhead could be avoided by embedding each planner into STeLLa. However, this is not desirable because it would entail losing the independence STeLLa has from the base planner.

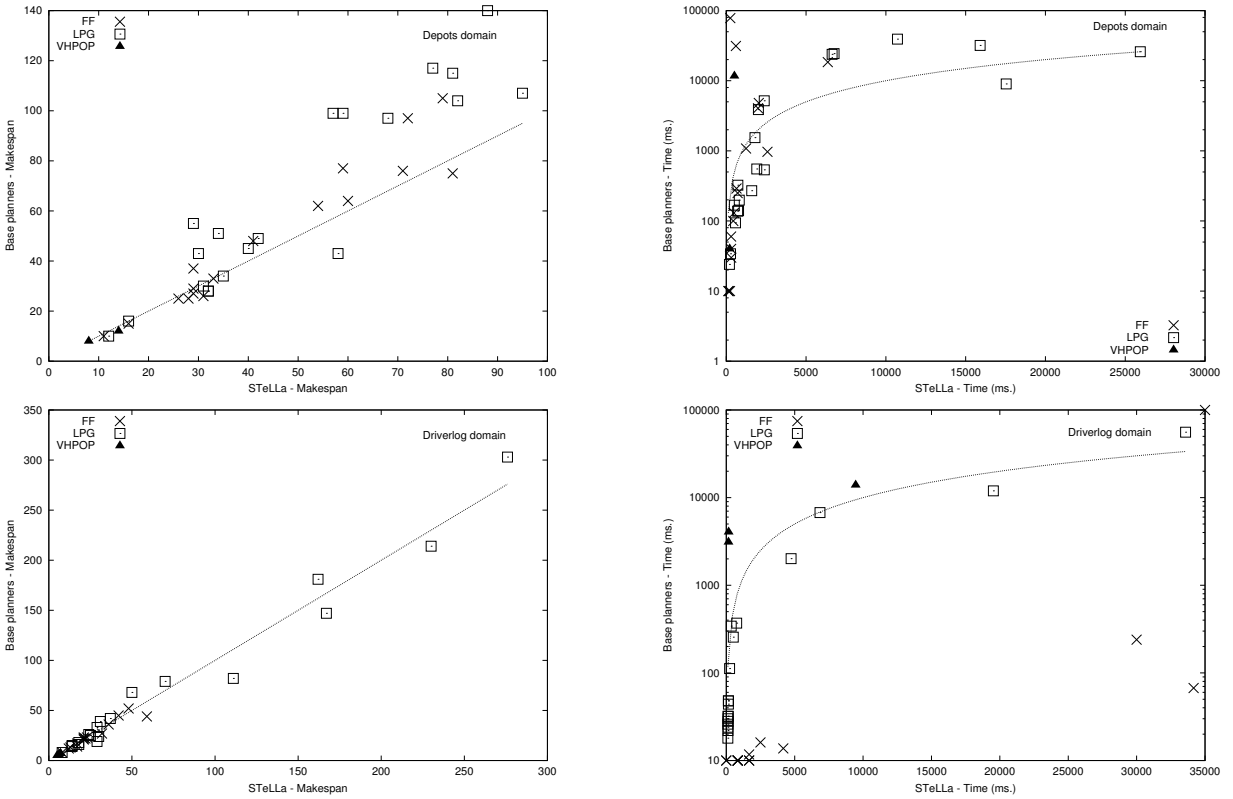

Fig. 10. Comparison between STeLLa and FF, LPG, and VHPOP in the depots and driverlog domains.

of this overhead.

As a conclusion, we can affirm that using STeLLa allows base planners to solve a greater number of instances with similar or even better quality than when solving the original problem. This is specially notable in domains with a high degree of interaction, such as the blocksworld and depots domains. On the other hand, its applicability is limited in terms of execution time. However, this limitation may be partially overcome by using a concurrent resolution of the subproblems, as Section 4 shows.

3.4.2. STeLLa-FF *vs.* FF-L *and* SGPlan

In this section, we present a comparison of STeLLa-FF planner with SGPlan and FF-L. These last two decomposition techniques also use FF to solve the obtained subproblems. **We have focused our comparison on decomposition techniques which use landmarks and FF as base planner**[13]. In these experiments, we restricted time consumption to 100 seconds. Table 5 shows the number of problems solved by each approach for each domain, Figures 12 to 14 show the performance in solving instances from IPC2 and IPC3. These charts can be interpreted in the same way as in the previous section.

FF-L is a decomposition technique based on the use of landmarks as explained in Section 2. This is the system STeLLa is based on and from whom it borrows some definitions (as said in section 3.1). FF-L computes a set of landmarks and a set of necessary, reasonable and obedient orders between them to obtain a landmark generation graph (LGG). The main difference between the LGG computed by FF-L and the LG computed by STeLLa is that STeLLa removes some incorrect necessary orders before calculating the reasonable and obedient orders which helps our system obtain more accurate IGs.

Once the LGG has been obtained, FF-L uses it to decompose the planning task into small chunks, which can be handed over to any planning algorithm (in [37] and [24], IPP, FF, and LPG were used). Unlike our approach (which computes con-

[13] This is the reason why we have not compared GRT with STeLLa.

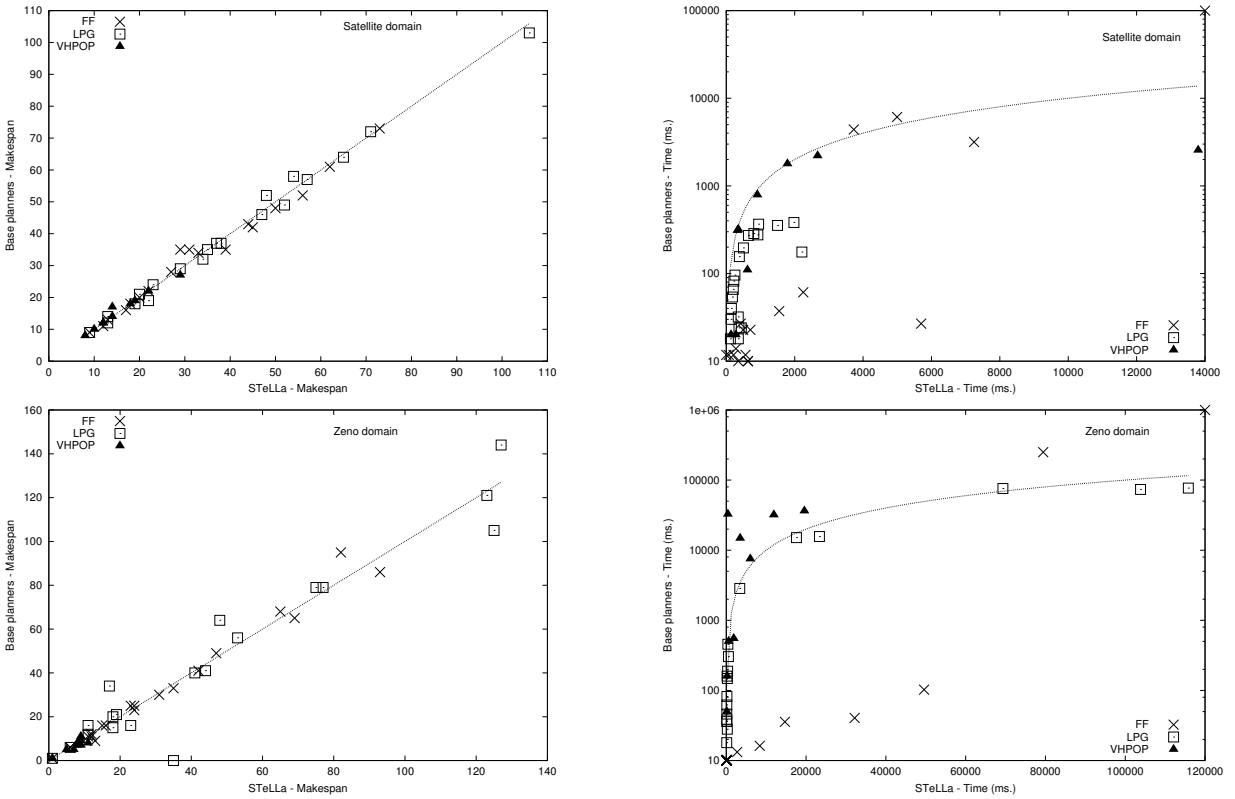

Fig. 11. Comparison between STeLLa and FF, LPG, and VHPOP in the satellite and zeno domains.

	Blocks	Elevator	Freecell	Logistics	Depots	Driverlog	Satellite	Zeno
Total	102	150	60	77	20	20	20	20
STeLLa-FF	102	150	48	77	20	15	18	20
FF-L	40	150	36	77	18	14	20	20
SGPlan	43	150	56	77	19	15	20	20

Table 5
Number of problems solved by STeLLa-FF, FF-L, and SGPlan for the domains of the IPC2 and IPC3 planning competitions.

junctive goals), FF-L builds disjunctive goals; it is said that a disjunctive goal is satisfied when at least one literal in the disjunctive goal has been achieved. This is the main difference between FF-L and STeLLa . Since the process to calculate these disjunctive goals does not reason about the convenience of including a literal in a certain goal, this makes the plans obtained by FF-L longer than the ones obtained by STeLLa-FF in domains such as elevator (Figure 12), logistics (Figure 13), depots (Figure 14) and zeno (Figure 15). Moreover, STeLLa-FF solves more problems than FF-L (Table 5). The reason behind these improvements is that our methods takes into account positive as well as additional negative interactions between literals in order to build the IGs. On the other hand, FF-L is slightly faster in general, mainly due to the overhead that the parsing of all the subproblems causes in STeLLa-FF.

SGPlan performs a problem decomposition at two levels: first, it partitions the problem into a set of subproblems with one subproblem for each top-level goal; second, each subproblem is then decomposed by landmarks. More precisely, SGPlan partitions the problem into n subproblems $G_1, \ldots, G_n$, one for each fact in the top-level goal. Then, it orders the subproblems according to three heuristics for partial ordering of subgoals. The first level is

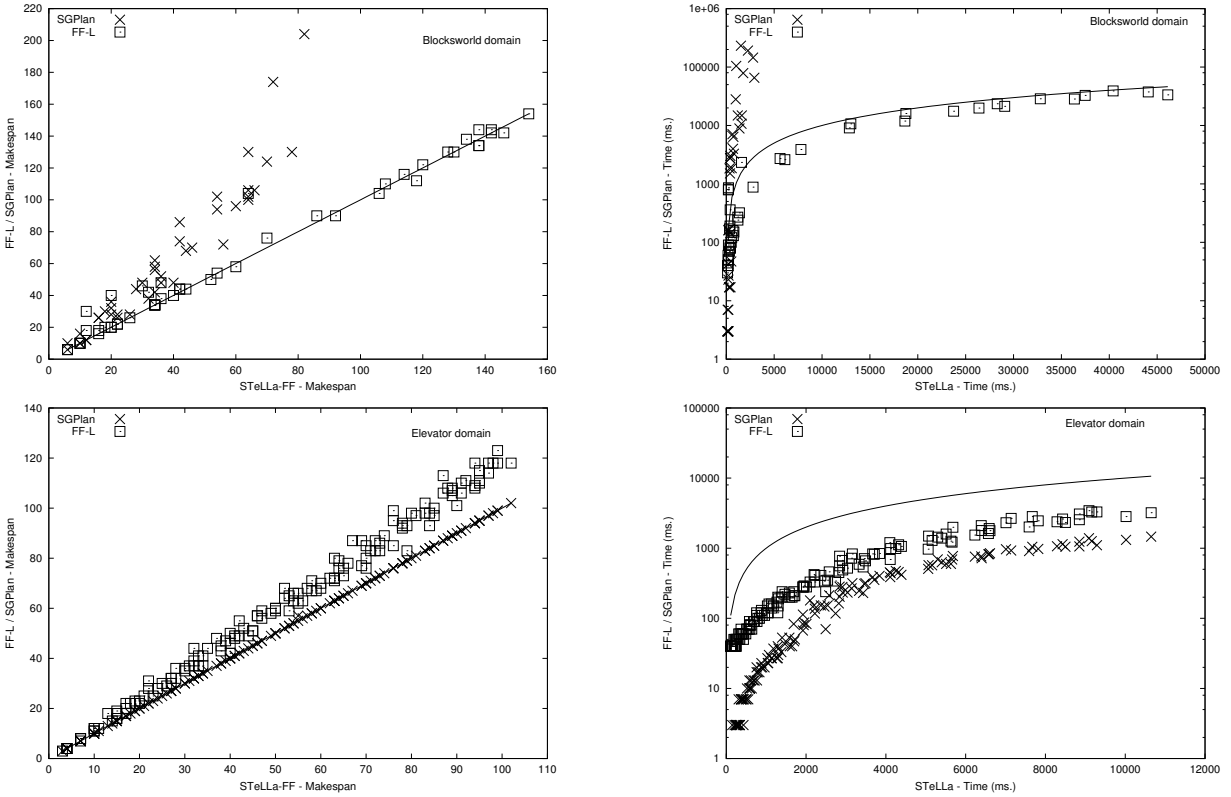

Fig. 12. Results of STeLLa-FF, SGPLAN, and FF-L in problems of the blocksworld and elevator domains.

called reasonable ordering, which was proposed in [29] (see above). The second and third level of ordering, called irrelevance and precondition ordering, respectively, are based on the idea of solving more difficult subgoals first. Once the subproblems have been ordered, SGPlan solves them. Sometimes the subproblems obtained after the first ordering level are still too large, so SGPlan applies a second level of decomposition by discovering landmarks [24] and ordering them according to what in [24] are called necessary orderings. This way, SGPlan solves each of these subproblems and obtains a plan for each G_i.

Therefore, at first glance, the decomposition method utilized by SGPlan is similar to the one used in STeLLa. However, the main difference between these two approaches lies in the order in which these landmarks are solved. SGPlan achieves each top-level goal separately, i.e. it ignores possible positive interacions. Therefore, the extracted landmarks do not contain this information. STeLLa, however, interleaves the computation of landmarks for the different top-level goals using the information provided by reasonable and obedient orders and active interferences. This explains why STeLLa outperformed SGPlan in highly interactive domains such as the blocksworld domain (Figure 12). In this domain STeLLa solved many more instances than SGPlan much faster and with better quality. A similar situation occured in the logistics (Figure 13) and depots (Figure 14) domains, whereas in the remaining domains, both approaches obtained solutions of similar quality, but SGPlan worked slightly faster.

As a conclusion we can affirm that there are two main differences between STeLLa and FF-L and SGPlan. Firstly, the technique for construction of the IGs and for handling cycles in the LG and, secondly, the architecture of STeLLa which allows different planners to be easily plugged into the framework.

4. Concurrent resolution

From the experiments in the previous section, one noticeable conclusion is that planners are able

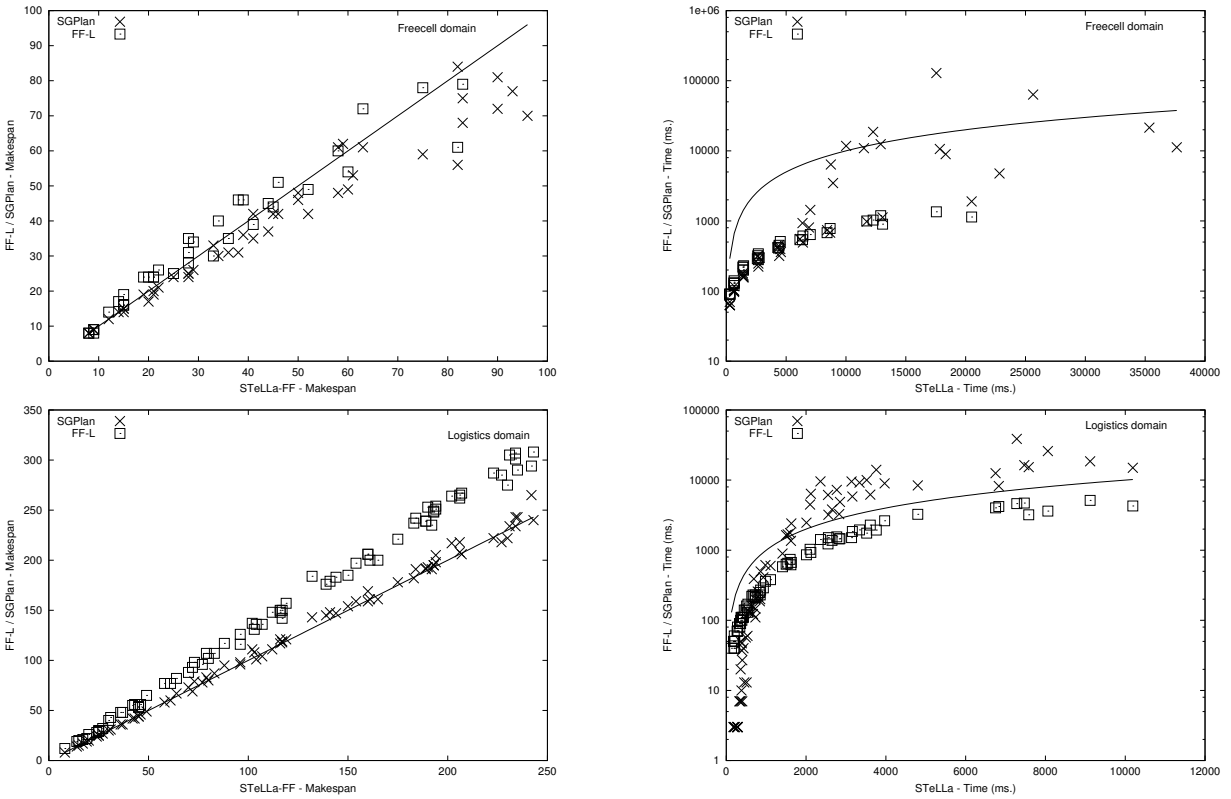

Fig. 13. Results of STeLLa-FF, SGPLAN, and FF-L in problems of the freecell and logistics domains.

to solve more problems when they use STeLLa to decompose planning problems. However, time performance gets worse (to the point that a planner with STeLLa only behaves better in some particular domains) because of (a) the time spent in computing the IGs and (b) the time spent in parsing each subproblem.

The IG construction process computes the goal sets of all subproblems before starting their resolution. Since all the IGs are known in advance, we might be able to come up with a method to calculate the final state reached by each subproblem and apply a concurrent resolution. A concurrent problem-solving technique may provide important time savings and improve STeLLa performance in the tested domains. This section details the technique we have developed in order to build the intermediate states before solving the subproblems. Our goal is to complete the goal sets to approximate the state that would result after solving each IG. This state is called *intermediate state*.

Definition 10 *An* **intermediate state (IS)** *is a complete description of the world that contains the literals from IG_i plus some additional literals that are necessary to complete the state reached after solving IG_i: $IS_i = IG_i \cup L$*

Ideally, IS_i would be the resulting state from solving problem $i-1$, which would be used as initial state for problem i. This formulation still implies a sequential resolution, so our goal is to calculate an approximation to IS_i without having to solve problem $i-1$. The process to approximate an intermediate state IS_i consists in calculating the set of literals L that makes IS_i be the closest consistent state to IG_i (according to IS_{i-1}). In order to do so, we use the concepts of **property space** and **state invariant**, which are explained in Section 4.1. Section 4.2 introduces the main ideas for constructing the ISs, and Section 4.4 shows how STeLLa's performance improves when using a concurrent resolution.

4.1. Property Spaces and State Invariants

In this section we repeat (sometimes in a reduced form) the definitions of property spaces and

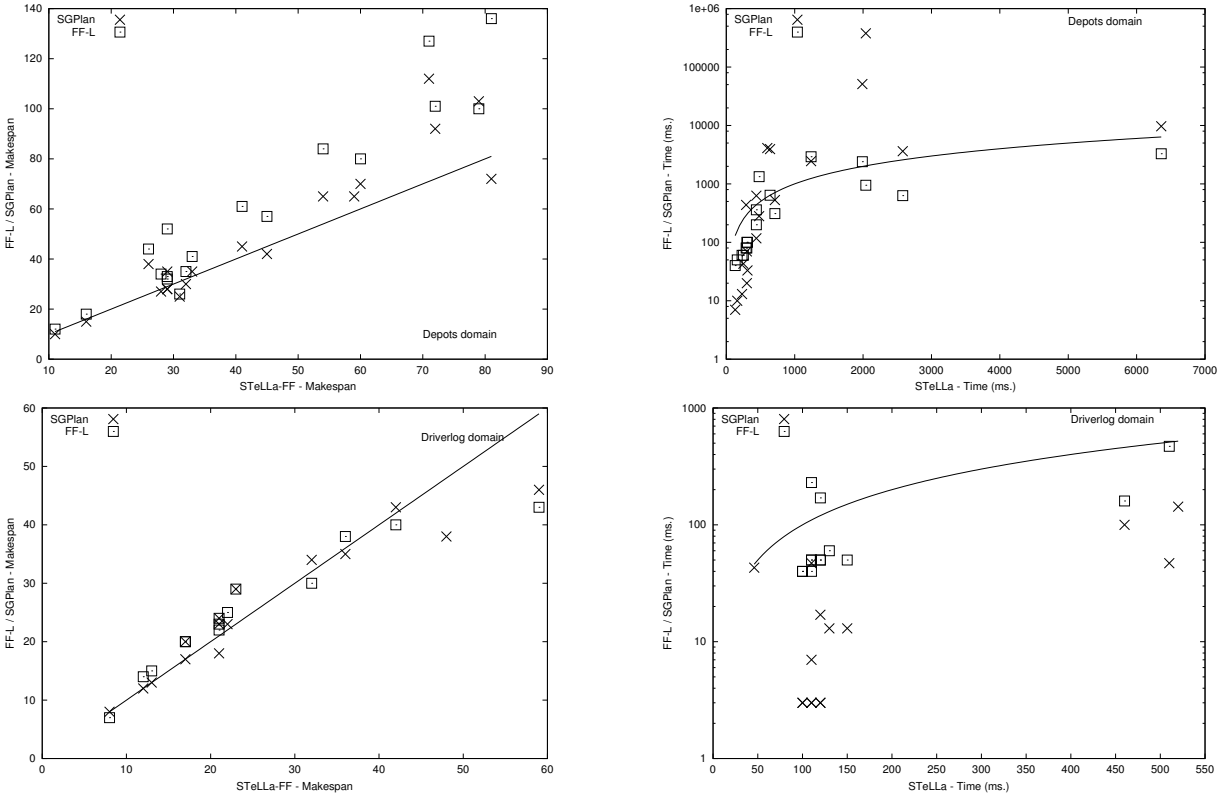

Fig. 14. Results of STeLLa-FF, SGPLAN, and FF-L in the depots and driverlog domains.

invariants given by Fox and Long [15], which make up the basis for our method to compute the intermediate states. A *property space*, a notion offered by TIM (Type Inference Module), defines which *properties* an object must necessarily hold in a particular problem state.

Definition 11 *A* **property** *is a predicate subscripted by a number between 1 and the arity of that predicate. Every predicate pr of arity n defines n properties:* $\{pr_1, pr_2, \ldots, pr_n\}$. *The set of all possible properties in a domain is denoted as* PR.

For example, in the depots domain, TIM builds two properties from the predicate lifting: lifting1 and lifting2.

TIM starts the analysis of the domain by representing the domain as a collection of finite-state machines (FSMs) with domain constants (objects) traversing the states within them. When two objects participate in identical FSMs, they can be considered as being of the same type. Figure 16 shows the FSMs for the type hoist in the depots domain. *Transition rules* represent the state transformations that comprise the FSMs traversed by the objects in the domain.

Definition 12 *A* **transition rule** *is an expression of the form:* $E \Rightarrow S \rightarrow F$, *in which the three components are bags of zero or more properties called* enablers, start *and* finish, *respectively. The set of all the possible transition rules in a domain is denoted as* TR. *A transition rule* R *specifies which properties an object gains or loses as a result of the application of an operator over a bag of properties* P:

$$result(P, R) = P - S \cup F \quad if \quad E, S \in P$$

As Figure 16 shows, any hoist which has property [lifting1] will gain property [available1] and will lose property [lifting1] when operators drop or load are applied. Also, hoists never lose property [at1] because no action can eliminate it.

Definition 13 *A* **property state** *(PS) is a bag of properties*[14].

[14] This concept is called *state* in [15], but we prefer to use *property state* to distinguish a bag of properties from a set of literals (state in the planning problem).

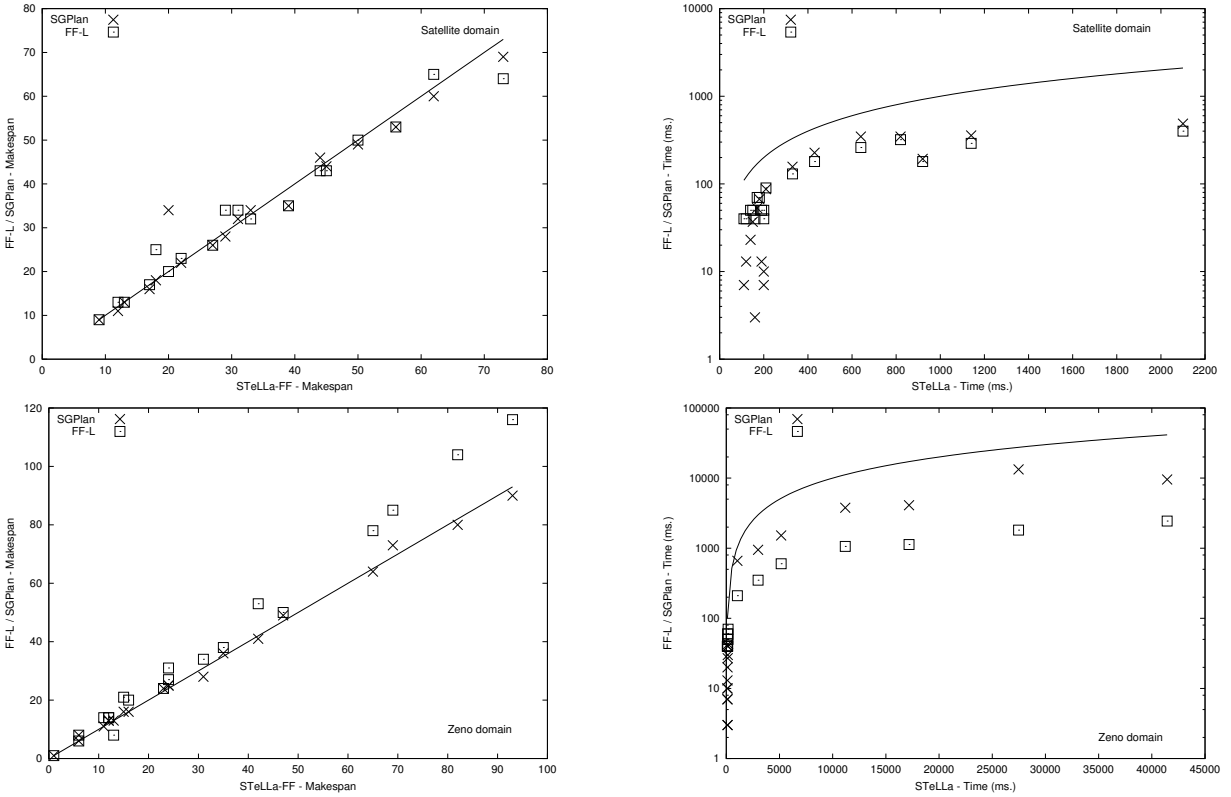

Fig. 15. Results of STeLLa-FF, SGPLAN, and FF-L in the satellite and zeno domains.

Definition 14 *A* **property space** *is a tuple of four components: a set of properties, a set of transition rules, a set of property states and a set of domain constants.*

TIM identifies the initial properties of individual objects and uses them to form property states of the objects in the property spaces. The initial states in a property space are then extended by the application of transition rules in the space to form complete sets of property states, which accounts for all the states that objects in that property space can possibly inhabit. Therefore, any object in a particular property space must have one (and only one) property state in each problem state. The list of property states in the property space for type hoist in the depots domain is: {{[at1], [lifting1]}, {[at1], [available1]}}. This means that a hoist is always at a location and it can either be available or it is lifting a crate. The following list defines the property states for a crate:

1. [clear1, on1, at1]: this PS indicates that, in a valid state, a crate is clear; it is on a pallet or on top of another crate; and it is at a distributor.
2. [on2, on1, at1]: this PS indicates that a crate has another crate on top of it; it is on a pallet or on top of another crate; and it is at a distributor.
3. [lifting2]: this PS indicates that a crate is being lifted by a hoist.
4. [in1]: this PS indicates that a crate is in a truck.

Finally, TIM uses the property spaces to determine state invariants that govern the behaviour of the domain and the objects in it. We are interested in *state membership invariants* and *uniqueness invariants* (invariants that characterize the uniqueness of state membership).

Definition 15 *A* **state membership invariant** *is an expression of the form:*

$$\forall x(Disjunct_1 \vee \ldots \vee Disjunct_n)$$

Only one state membership invariant is constructed for each property space. Each property

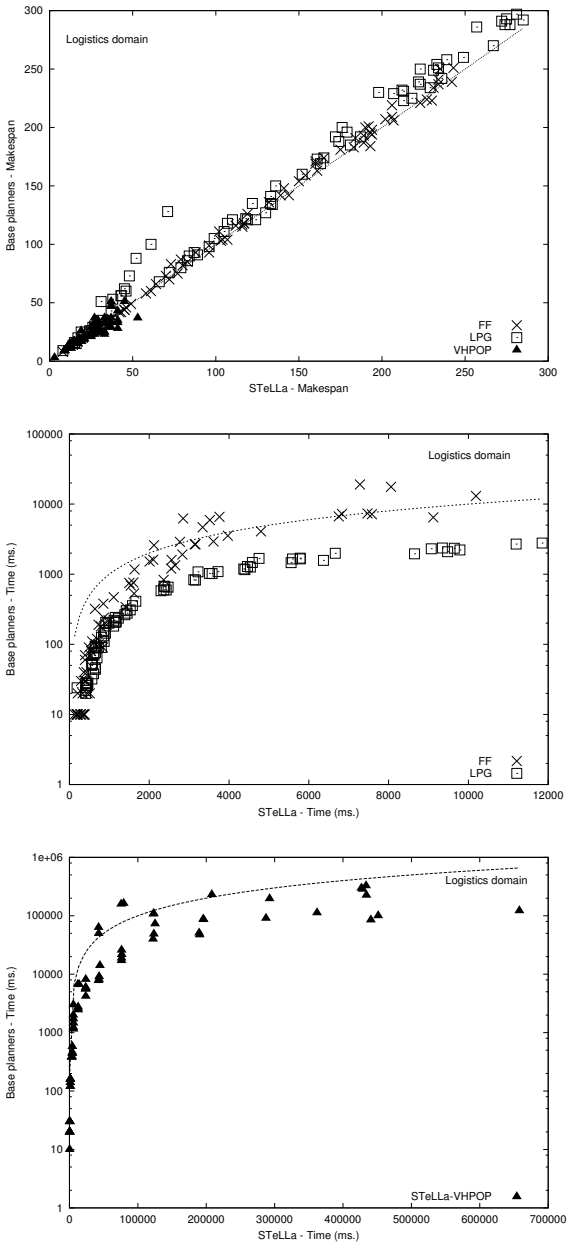

Fig. 9. Comparison between STeLLa and FF, LPG, and VHPOP in the logistics domain.

state of the property space is used to build a single disjunct. If the property state contains m properties denoted as $P^1 \dots P^m$, TIM builds the following disjunct (assuming that x is in the first position of the predicate):

$\exists \overline{y}_1 \dots \overline{y}_m (P^1(x, \overline{y}_1) \wedge P^2(x, \overline{y}_2) \wedge \dots \wedge P^m(x, \overline{y}_m))$

where $\overline{y}_i$ are vectors containing $n-1$ values. That is, a state membership invariant states that an object associated with a property space satisfies at least one property state of that property space.

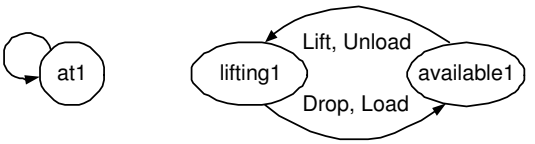

Fig. 16. Transition rule for hoists

For example, the state membership invariant for the type crate is:

$\forall x \in \text{crate} : ((clear(x) \wedge \exists y \in \text{crate} \cup \text{pallet} : on(x,y) \wedge \exists y \in \text{location} : at(x,y)) \bigvee (\exists y \in \text{crate} \cup \text{pallet} : on(x,y) \wedge \exists y \in \text{crate} : on(y,x) \wedge \exists y \in \text{location} : at(x,y)) \bigvee (\exists y \in \text{hoist} : lifting(y,x)) \bigvee (\exists y \in \text{truck} : in(x,y)))$

It is easy to see that this invariant is a representation of the PS list for the type crate enumerated above in first order logic.

Definition 16 *A* **uniqueness invariant** *states that each object associated with a property space satisfies no more than one property state of that property space. A uniqueness invariant is defined for every pair of property states P,Q (assuming that x is in the first position of the predicate):*

$$\forall x \neg((\exists \overline{y}_1 \dots \overline{y}_n (P^1(x, \overline{y}_1) \wedge \dots \wedge P^n(x, \overline{y}_n))) \wedge (\exists \overline{y}_1 \dots \overline{y}_m (Q^1(x, \overline{y}_1) \wedge \dots \wedge Q^m(x, \overline{y}_m))))$$

where $\overline{y}_i$ are vectors containing $n-1$ values.

For example, one of the uniqueness invariants for the type crate is:

$\forall x \in \text{crate} : \neg((clear(x) \wedge \exists y \in \text{crate} \cup \text{pallet} : on(x,y) \wedge \exists y \in \text{location} : at(x,y)) \bigwedge (\exists y \in \text{hoist} : lifting(y,x)))$

This invariant indicates that there is no valid state if a crate is on a surface (pallet or another crate) and being lifted by a hoist at the same time. A set of invariants in the same form is defined to prevent a crate from (partially or completely) satisfying two PSs at the same time.

As [15] details, the process used to compute the property spaces guarantees that all the reachable states will be in the appropriate property space. In addition, the complexity of this process has been proved experimentally to be linear with the number of operators and constants in the domain and with the size of the initial state of the particular problem.

4.2. Intermediate states

In general, given a subproblem $P_i = \langle \mathcal{A}, IS_{i-1}, IG_i \rangle$, IS_i will be known after solving P_i. Our aim is to eliminate this dependency between subproblems and solve all P_i concurrently. Therefore, we need to compute each IS_i in advance. The input of this process will be P_i and the property spaces and invariants discovered for the corresponding domain.

As an initial approximation, IS_i will contain the literals in IG_i, since literals in the goal set will appear in the state reached after solving P_i. Additionally, we could add the literals in IS_{i-1} that are consistent with literals in IG_i. However, this approximation of IS_i might likely contain incorrect literals. Likewise, there could be missing literals that should appear in IS_i. Next, we state the formal definitions for a correct intermediate state.

Definition 17 *An intermediate state IS_i is said to be* **correct** *when both the state membership invariants and the uniqueness invariants are satisfied by all of the objects in the domain.*

This affirmation relies on the fact that a state membership invariant represents all the possible states that an object of a certain type can reach, and all the uniqueness invariants related with that type indicate that the object must be at only one of those states. The combination of these invariants will result in this object being in a correct state provided that both the state membership and the uniqueness invariants inferred by TIM are correct. This has been proved in [15].

The problem of building a correct IS_i can be formulated as the process of building a correct state S_o for each object in the planning problem, that is: $IS_i = \bigcup_{\forall o} S_o$. Each S_o will contain the literals in IG_i referring to o. The remainder of this section is focused on explaining how S_o is computed for a particular object o. First, we introduce some basic definitions.

Definition 18 *An object o of a planning problem is said to* **belong** *to a literal l (and denoted as belongs(o,l)) when it appears as an argument of this literal.*

Definition 19 *Given an object o and a property $p = pr_i$, such that $p \in o$, we say that p can be* **instantiated** *with a valid literal $l = pr'(a_1, \ldots, a_n)$, which is denoted as $p \rhd l$, if $pr = pr'$ and $belongs(o, l)$.*

For example, given the object Hoist0 and the literal (lifting Hoist0 Crate0), we can affirm that *belongs(*Hoist0,(lifting Hoist0 Crate0)*)* is true. On the other hand, given the same object and the property lifting1, lifting1 can be instantiated with literal (lifting Hoist0 Crate0).

The process to build S_o for a particular object o is based on the property spaces discovered for the corresponding domain. Each object must fulfill the state membership invariant at each state. This means that the properties an object has in a particular state will coincide with a property state (and only one, due to the uniqueness invariants). Therefore, if we find which PS is closer to the state that o must have in IS_i, we can complete S_o with an instantiation of the remaining properties, that is, those that are not present in IG_i. Therefore, in the first step, we study the properties that the object o has in IS_{i-1} and we determine which properties the object o will gain and lose in IS_i.

Definition 20 *Given IS_{i-1} and IG_i and a property state PS_m, we define the following sets for any object o:*

- $CurrProp_o = \{p \in PR / \exists l \in IS_{i-1} : (p \rhd l \wedge belongs(o,l))\}$*, this set will contain those properties that o currently has.*
- $AddProp_o = \{p \in PR / \exists l \in IG_i : (p \rhd l \wedge belongs(o,l))\}$*, this set represents the properties that o must have in IS_i.*
- $DelProp_o = \{p \in PR / \exists l \in IS_{i-1} : (\exists l' \in IG_i : inconsistent(l,l')) \wedge p \rhd l \wedge belongs(o,l)\}$*, this set indicates which properties of o will not be in IS_i.*

The next step consists of **selecting a property state** PS_o among all the PSs offered by the property space associated with the type of o. This PS_o will be the PS that is the most compatible with the properties contained in $CurrProp_o$, $AddProp_o$ and $DelProp_o$, since these sets denote all the information we know about o. Namely, we apply the following definition to determine which PS is closer to the state that o will have in IS_i.

Definition 21 *Given two property states PS_i and PS_j, PS_i is* **closer** *to the state of an object o in IS_i than PS_j if any of the following conditions hold:*

(a) PS_i is closer than PS_j if there is a transition rule that if applied to the current properties of o generates a set of properties that

contains PS_i and we cannot find such a transition rule for PS_j:
$\exists R_i \in TR : PS_i \subseteq result(CurrProp_o, R_i) \wedge \neg\exists R_j \in TR : PS_j \subseteq result(CurrProp_o, R_j)$

(b) PS_i is closer to the state of o in IS_i if all the properties that must be added belong to PS_i but not to PS_j:
$\forall p \in AddProp_o : p \in PS_i \wedge \exists p \in AddProp_o : p \notin PS_j$

(c) PS_i is closer than PS_j if it does not contain any deleted property and PS_j does:
$\forall p \in DelProp_o : p \notin PS_i \wedge \exists p \in DelProp_o : p \in PS_j$

(d) PS_i will be closer than PS_j if the number of properties in $CurrProp_o$ and $AddProp_o$ that do not belong to PS_i is smaller than the number of properties in these sets that do not belong to PS_j:
$|\{p \in PS_i : p \notin CurrProp_o \cup AddProp_o\}| < |\{p \in PS_j : p \notin CurrProp_o \cup AddProp_o\}|$

Given PS_o, the most compatible PS with properties in $CurrProp_o$, $AddProp_o$, and $DelProp_o$, the final step for building S_o is to **find a set of literals such that each of them instantiates one property in** PS_o. The properties in $AddProp_o$ will be directly instantiated with the literals in IG_i that refer to o; we call this set $AddLits_o$ and it is defined as: $AddLits_o = \{l \in IG_i / belongs(o, l)\}$. The remaining properties in PS_o will be instantiated with the side-effects of the actions that add the literals in $AddLits_o$. The objective is to select the most appropriate action (among those that add the literals in $AddLits_o$) to be able to instantiate as many remaining properties as possible. In case there are several candidate actions, we select those actions that match the greatest number of remaining properties. From this new set, we give higher priority to those actions that have more preconditions supported in IS_{i-1} that are not related to the object o. The idea behind is to first select an action that assumes that the states of the rest of objects remains unaltered. If there exists an action that is completely applicable in IS_{i-1}, we select that action; otherwise, we assume the literals in $AddLits_o$ have been achieved through the application of a sequence of actions (particularly by the last action of the sequence) and, therefore, none of these last actions will be totally applicable in IS_{i-1}. In this latter case, an action is arbitrarily selected. Once an action is selected, the remaining properties of PS_o will be instantiated with the corresponding arguments of the instantiated action effects.

The process for building S_o guarantees that S_o satisfies the corresponding state membership and uniqueness invariants, because we select one (and only one) PS from all the possible PSs and then only one literal per property is included in S_o. Therefore, an IS_i which is built as the union of all S_o (one for each object) will be a correct state.

It is important to remark that S_o is built, in principle, only for all the objects that appear in IG_i. However, in some cases, the state built for an object o may imply changing the state of another object o'. Then, the new state of o', $S_{o'}$ is also computed. **When the state of one object has been computed, it is used as a basis for computing the state of the remaining objects so that the final IS_i is completely consistent.**

We now show an example of computing S_o in the depots problem in Figure 1 for the object Crate0. Let IS_3 contain the following literals, among others: {(lifting Hoist2 Crate0), (at Hoist2 Distributor0), (clear Pallet2), (at Pallet2 Distributor0)}. Let literal (on Crate0 Pallet2) belong to IG_4. We first calculate the following sets of properties: $CurrProp_o$ = {[lifting2]}, $AddProp_o$ = {[on1]}, and $DelProp_o$ = {[lifting2]}. Taking into account the property space for the type crate shown in Section 4.1, the closest PS will be [clear1, on1, at1] because:

1. the reachable PSs from [lifting2] with a transtion rule are [in1] and [clear1, on1, at1]
2. only [clear1, on1, at1] contains all the properties in $AddProp_o$.

Once the most compatible PS has been selected, the last step is to instantiate these properties with literals. Property [on1] is instantiated with (on Crate0 Pallet2), which belongs to IG_4, and property [clear1] is directly instantiated as it only has one argument, which forms the literal (clear Crate0). Then, there is only one remaining property, [at1], which has to be instantiated with a literal added by an action that has (on Crate0 Pallet2) and (clear Crate0) as positive effects. There are a number of actions that add these literals: (drop Hoist0 Crate0 Pallet2 Depot0); (drop Hoist1 Crate0 Pallet2 Depot0); (drop Hoist2 Crate0 Pallet2 Distributor0); and (drop Hoist3 Crate0 Pallet2 Distributor1)). However, only the preconditions of (drop Hoist2 Crate0 Pallet2 Distributor0) that are not re-

lated to Crate0 are supported in IS_3. This implies that [at1] will be instantiated with (at Crate0 Distributor0).

Once all the ISs have been computed, the problems can be solved concurrently by any STRIPS planner. Then, the solutions obtained for these problems are concatenated to build the final solution plan. As these problems are independent, they can be solved simultaneously in a multiprocessor system.

4.3. Correctness

Given that ISs represent correct states (Definition 17), a plan $P_i = \langle \mathcal{A}, IS_{i-1}, IG_i \rangle$ to solve a problem will be a correct plan (see Section 3.3). However, it may be the case that after concurrently solving the subproblems, the actual state reached by each subproblem (RS) differs from the computed IS used as initial state for the subsequent subproblem. We say that RS and IS differ when there exists at least one object with a different property state in each state or with the same property state instantiated with different arguments. The difference between a reached state RS and the calculated IS is due to two factors:

1. The configuration of those objects that have not been managed when building the IS (that is, those objects not involved at all in the process of building the IS) remain the same as in IS_{i-1}. However, it might happen that the collateral effects (side-effects) of some action in the sequence to achieve IG_i from IS_{i-1} would change the state of those objects. In this case, the corresponding IS and RS will not coincide.
2. A special case of the above situation is found when a set of objects of the same type can be used for the same purpose (for example, having several trucks for transporting crates in the depots domain). The IG does not specify which object to use to achieve its goals and, therefore, the state of those objects will remain the same as in IS_{i-1}. However, since the RS will reflect the actual state of these objects, this may imply a difference between the RS and the IS.

When the reached state and the computed IS are not exactly the same state, subplans cannot be concatenated because they do not form a valid plan. One way to solve this difficulty is to replace the goal set of each subproblem by the corresponding IS, i.e., build subproblems as $P_i = \langle \mathcal{A}, IS_{i-1}, IS_i \rangle$. These problems are more costly to solve since the IG is replaced by a complete world description and solution plans are, in general, unnecessarily long.

It is important to note that these situations in which the reached state and the IS do not coincide are not frequent, so introducing a general overload in the problem resolution does not seem to be a good idea. Another solution is to repair the plan when necessary (the solution adopted in the experiments described in Section 4.4). When we detect a difference between the RS and the IS, we apply a repairing process to calculate the necessary actions to reach IS from RS. The concatenation of all the subplans plus these "repairing" subplans will form a valid plan and, therefore, the correctness exhibited in the sequential resolution is preserved in the concurrent solving.

Obviously, this second solution will also produce long plans, but only when a difference between RS and IS is detected. In this case, we could use rewriting techniques to produce better quality plans. This is part of our ongoing work.

As for completeness, we have not experimentally observed a difference with respect to the sequential resolution; that is, the intermediate states produced with our algorithm do not prevent the planner from obtaining the solutions to problems that would be solved with a sequential resolution.

4.4. Experiments

In this section, we present a comparison between the sequential (hereafter STeLLa-planner-Seq) and concurrent (hereafter STeLLa-planner-Conc) resolution of the subproblems obtained when applying STeLLa for domains where the sequential resolution proved to be useful (blocksworld, elevator, logistics and depots). We also compare STeLLa-planner-Conc with the performance of the base planners (FF, LPG, and VHPOP), SGPlan and FF-L.

The results we present in this section attempt to demonstrate that a concurrent resolution may imply important time savings with respect to the sequential resolution. **We do not pay attention to quality issues as the plans we obtain when applying the concurrent resolution are almost identi-**

cal to those obtained when applying the sequential resolution. Time consumption in concurrent resolution has been approximated as follows: the time required to decompose the problem and to combine the obtained subplans (T_{decomp}) plus the maximum between (1) the time used to solve all problems sequentially ($T_{problems}$) divided by the number of processors (n) and (2) the time used to solve the largest problem ($T_{max\ pr}$).

$$Time = T_{decomp} + Max\left(\frac{T_{problems}}{n}, T_{max\ pr}\right)$$

We assume that each problem is solved in a processor, so problems that are decomposed into n problems need n processors as maximum. This calculation provides an idea of the minimum time required to concurrently solve a partitioned problem and help determine in which cases it is worth using a concurrent resolution.

Figure 17 shows a comparison between STeLLa-planner-Seq and STeLLa-planner-Conc using FF, LPG, and VHPOP as base planners. These charts are interpreted the same way as those in Section 3.4, where the X and Y axes represent the results obtained with STeLLa-planner-Seq and STeLLa-planner-Conc, respectively. It can be observed that the concurrent resolution is, in most cases, better than the sequential resolution. This means that the concurrent resolution of all the subproblems makes up for the overload caused by the IS calculation and the repairing process (if necessary).

Table 6 shows a comparison between FF, LPG, and VHPOP when solving the original problem and when they are used as base planners in the sequential and in the concurrent resolution. Table 7 shows a comparison between STeLLa-FF (sequential and concurrent resolution), SGPlan and FF-L. In all cases, values represent the average time used by all configurations to solve the same set of problems (that is, we only took into account those problems solved by all configurations). In general terms, STeLLa-planner-Conc shortens the difference between STeLLa-planner-Seq and the corresponding base planner, as expected from the results shown in Figure 17. That is to say, in the domains where STeLLa-planner-Seq obtained a worse result than the base planner, STeLLa-planner-Conc still did not perform better although this difference was much smaller. However, there are some notable cases such as the logistics domain, where only STeLLa-FF-Conc obtained a better performance than FF (and the same occured with STeLLa-VHPOP-Conc and VHPOP). It is also noticeable that the difference in the depots domain between STeLLa-planner-Seq and STeLLa-planner-Conc was due to two instances for which STeLLa spendt an unusual amount of time (Figure 17).

In conclusion, we can affirm that the concurrent resolution allows STeLLa to obtain comparable results in terms of performance with several state-of-the-art planners. This, together with the good results obtained in terms of solution quality that are shown in Section 3.4, makes STeLLa an excellent technique for decomposing planning problems.

5. Conclusions and further work

Decomposition techniques have not been widely used in planning because of the difficulty of decomposing a problem into separate and independent subproblems. In highly interactive domains, partitioned problems cannot be solved without devoting great effort to gathering together the solution plans of the different subproblems. In this paper, we have presented a new decomposition technique for STRIPS domains, STeLLa, which does not have the same difficulties as other decomposition techniques. STeLLa applies a chronological decomposition and obtains a sequence of problems that can be solved without interleaving the goals that arise in the separate subproblems. In fact, the inherent interactions among different subgoals are used by STeLLa to decompose the problem. This technique is based on the notion of landmark. Landmarks are ordered and grouped together into different ordered intermediate goals, which correspond to the goal sets of the new problems. Under this sequential resolution scheme, the experiments give rise to two important conclusions: a) hard problems that were not solved by a specific planner became affordable when the decomposition technique was applied; as a whole, planners were able to solve more problems when using STeLLa and b) the quality of the solution plans was not lower in the partitioned problem; in many problem instances, it was even improved.

Once the benefits of applying STeLLa were experimentally proven, we proceeded to exploit the technique by providing a concurrent resolution scheme. The idea is to complete the initial state of each problem, which is inferred from the state that

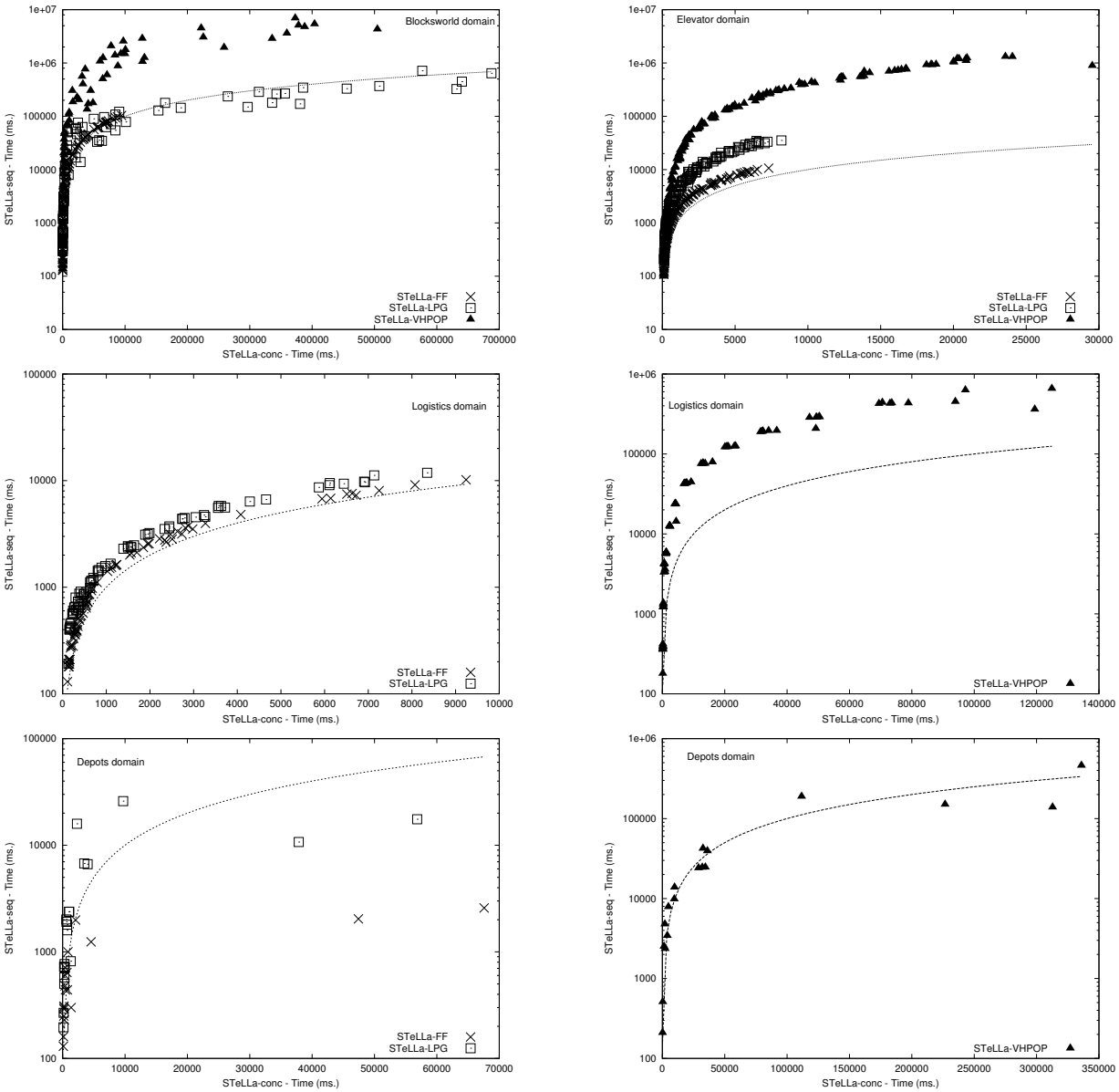

Fig. 17. Comparison between STeLLa when solving the subproblems sequentially and concurrently with FF, LPG, and VHPOP as base planners in the blocksworld, elevator, logistics and depots domains.

would be reached after solving the preceding goal set. Although concurrent STeLLa does not outperform other state-of-the-art planners (due to the parsing time devoted to each subproblem, among other reasons), important time savings were obtained with respect to the sequential mechanism.

As a conclusion we can affirm that there are several differences between STeLLa and other decomposition approaches. Firstly, the technique for construction of the IGs and for handling cycles in the LG; secondly, the architecture of STeLLa which allows different planners to be easily plugged into the framework and, thirdly, the use of state completion technique to support concurrent resolution of goals.

For future work, we are working on two different lines of research. One of them is to work on a relaxation of the landmark definition to obtain not only literals that appear in all solution plans, but also literals that appear in plans that are consid-

Planner	Blocksworld	Elevator	Logistics	Depots
FF	2.09	0.08	1.91	8.23
STeLLa-FF-Seq	16.27	2.51	1.92	1.1
STeLLa-FF-Conc	14.33	1.58	1.6	7.46
LPG	19.54	0.14	0.68	8.36
STeLLa-LPG-Seq	11.65	7.57	2.76	5.01
STeLLa-LPG-Conc	5.2	1.76	1.77	6.09
VHPOP	12.29	7.8	47.14	5.85
STeLLa-VHPOP-Seq	0.19	2.59	95.88	0.36
STeLLa-VHPOP-Conc	0.12	0.34	17.93	0.25

Table 6

Average time (secs.) required to solve the problems of each domain by using FF, LPG, and VHPOP and when these planners are combined with STeLLa solving the subproblems sequentially and concurrently.

Planner	Blocksworld	Elevator	Logistics	Depots
SGPlan	22.67	0.25	3.75	24.69
STeLLa-FF-Seq	0.71	2.51	1.92	1.1
STeLLa-FF-Conc	0.48	1.58	1.6	7.08
FF-L	6.95	0.62	0.96	0.61
STeLLa-FF-Seq	8.9	2.51	1.92	1.05
STeLLa-FF-Conc	7.74	1.58	1.6	7.47

Table 7

Average time (secs.) required by STeLLa-FF, SGPlan, and FF-L to solve the problems of each domain.

ered to be of good quality. This will bring the advantage of being able to obtain intermediate goals for problems that are not currently decomposable. The other line of research is to apply STeLLa decomposition to temporal planning problems.

Acknowledgements

This work has been partially funded by the Spanish government CICYT project TIC2002-04146-C05-04, by the Valencian government project GV04A-388 and by the UPV project 20020681.

The authors wish to thank Maria Fox and Derek Long for providing the TIM API and also for modifying it to facilitate the access of STeLLa to property states functions. We are also very thankful to Joerg Hoffmann and Julie Porteous for the ideas introduced in the landmarks theory which have been later used to develop our decomposition technique. We finally wish to thank the anonymous reviewers for their comments which were very helpful to improve the paper.

References

[1] J.L. Ambite and C.A. Knoblock. Planning by rewriting: efficiently generating high-quality plans. In *14th National Conference on AI*, 1997.

[2] F. Bacchus. AIPS-2000 competition results. Technical report, University of Toronto, 2000. http://www.cs.toronto.edu/aips2000/.

[3] F. Bacchus and Q. Yang. Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence*, 71:43–100, 1994.

[4] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

[5] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *5th European Conf. on Planning (ECP-99)*, 1999.

[6] B. Clement and E. Durfee. Top-down search for coordinating the hierarchical plans of multiple agents. In *Third International Conference on Autonomous Agents*, pages 252–299, 1999.

[7] D. Corkill. Hierarchical planning in a distributed environment. In *6th International Joint Conference on Artificial Intelligence*, 1979.

[8] K. Currie and A. Tate. O-plan: the open system architecture. *Artificial Intelligence*, 52:49–86, 1991.

[9] S. Das, P. Gonsalves, R. Krikorian, and W. Truszkowski. Multi-agent planning and scheduling environment for enhanced spacecraft autonomy. In *5th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, 1999.

[10] K. Decker and V. Lesser. Generalizing the partial global planning algorithm. *International Journal of Intelligent and Cooperative Information Systems*, 1(2):319–346, 1992.

[11] K. Decker, V. Lesser, and M.V.Nagendra Prasad. Macron: an architecture for multi-agent cooperative information gathering. In *Conference on Information and Knowledge Management. Workshop on Intelligent Information Agents*, 1995.

[12] M.E. Desjardins and M.J. Wolverton. Coordinating planning activity and information flow in a distributed planning system. *AI Magazine*, pages 45–53, 1999.

[13] J. Dix, H. Munoz-Avila, D.Nau, and L. Zhang. IMPACTing SHOP: Putting an AI planner into a multi-agent environment. *Annals of Mathematics and AI*, 2002.

[14] E. Durfee and V. Lesser. Partial global planning: a coordination framework for distributed hypothesis formation. *IEEE Transactions on Systems, Man and Cybernetics*, 1(1):63–83, 1991.

[15] M. Fox and D. Long. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*, 9:367–421, 1998.

[16] M. Fox and D. Long. Hybrid STAN: Identifying and managing combinatorial sub-problems in planning. In *Proceedings of IJCAI'01*, pages 445–452, 2001.

[17] M. Fox and D. Long. Domains and results of the third international planning competition, 2002. http://www.dur.ac.uk/d.p.long/competition.html.

[18] M. R. Garey and D. S. Johnson. *Computers and Intractability: a guide to the theory of NP-completeness.* W. H. Freeman and Company, San Francisco, 1979.

[19] A. Gerevini and L.K. Schubert. Discovering state constraints in DISCOPLAN: Some new results. In *17th Nat. Conf. on Artificial Intelligence (AAAI'2000)*, 2000.

[20] A. Gerevini and I. Serina. LPG: a planner based on local search for planning graphs. In *AIPS'02*. AAAI Press, 2002.

[21] J. Hoffmann. Extending FF to numerical state variables. In *ECAI'02*, pages 571–575. IOS Press, Amsterdam, 2002.

[22] J. Hoffmann. Local search topology in planning benchmarks: A theoretical analysis. In *AIPS'02*, 2002.

[23] J. Hoffmann and B.Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[24] J. Hoffmann, J. Porteous, and L. Sebastia. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–287, 2004.

[25] M. Iwen and A. Mali. Automatic problem decomposition for distributed planning. In *International Conference on Artificial Intelligence (IC-AI'2002)*, volume 1, pages 411–417, 2002.

[26] M. Iwen and A. Mali. Distributed graphplan. In *14th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'2002)*, pages 138–145, 2002.

[27] K. Knight. Are many reactive agents better than a few deliberative ones? In *13th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 432–437, 1993.

[28] C. A. Knoblock. *Generating Abstraction Hierarchies - An automated approach to reducing search in planning.* Kluwer, Dordrecht, Netherlands, 1993.

[29] J. Koehler and J. Hoffmann. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *Journal of Artificial Intelligence Research*, 12, 2000.

[30] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an ADL subset. In *ECP'97*. Springer LNAI, 1997.

[31] R. Korf. Planning as search: a quantitative approach. *Artificial Intelligence*, 33:65–85, 1985.

[32] A. Lansky. Scope and abstraction: Two criteria for localized planning. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1612–1619, 1995.

[33] D. Long and M. Fox. Efficient implementation of the plan graph in STAN. *Journal of Artificial Intelligence Research*, 10:87–115, 1999.

[34] D. Long and M. Fox. Automatic synthesis and use of generic types in planning. In *5th Conference on Artificial Intelligence Planning Systems*, pages 196–205, 2000.

[35] A. Mali and S. Kambhampati. Distributed planning. *The Encyclopaedia of Distributed Computing*, 1999.

[36] D. Nau, Y. Cao, A. Lotem, and H. Munoz-Avila. Shop: Simple hierarchical ordered planner. In *16th International Joint Conference on AI*, pages 968–975, 1999.

[37] J. Porteous, L. Sebastia, and J. Hoffmann. On the extraction, ordering, and usage of landmarks in planning. In *Recent Advances in AI Planning. ECP'01*. Springer Verlag, 2001.

[38] I. Refanidis and I. Vlahavas. The GRT planning system: backward heuristic construction in forward state-space planning. *Journal of Artificial Intelligence Research*, pages 15:115–161, 2001.

[39] E. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, pages 5:115–135, 1974.

[40] B. Srivastava. Realplan: Decoupling causal and resource reasoning in planning. In *17th National Conference on AI*, pages 812–818, 2000.

[41] B. W. Wah and Y. Chen. Constrained partitioning in penalty formulations for solving temporal planning problems. *Artificial Intelligence*, 2004 - to appear.

[42] D. E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann Publishers Inc., 1988.

[43] B. W. Wah Y. Chen and C. Hsu. Temporal planning using subgoal partitioning and resolution in SGPlan. *Journal of Artificial Intelligence Research*, 2005 - to appear.

[44] Q. Yang. *Intelligent Planning. A Descomposition and Abstraction Based Approach.* Springer-Verlag, Berlin, Heidelberg, 1997.

[45] H. Younes and R. Simmons. On the role of ground actions in refinement planning. In *Proceedings of the Sixth Int. Conference on AI Planning and Scheduling (AIPS'02).* AAAI Press, 2002.