Towards compositional automated planning

Endre Erős[1], Martin Dahl[1], Petter Falkman[1] and Kristofer Bengtsson[1]

***Abstract*—The development of efficient propositional satisfiability problem solving algorithms (SAT solvers) in the past two decades has made automated planning using SAT-solvers an established AI planning approach. Modern SAT solvers can accommodate a wide variety of planning problems with a large number of variables. However, fast computing of reasonably long plans proves challenging for planning as satisfiability. In order to address this challenge, we present a compositional approach based on abstraction refinement that iteratively generates, solves and composes partial solutions from a parameterized planning problem. We show that this approach decomposes the monolithic planning problem into smaller problems and thus significantly speeds up plan calculation, at least for a class of tested planning problems.**

***Index Terms*—automated planning, planning as satisfiability, online planning, compositional planning, artificial intelligence, abstraction refinement, SAT solvers, intelligent automation.**

I. Introduction

Increased industrial competitiveness requires fundamental changes in automation. As companies introduce collaborative, intelligent and flexible systems into production to meet the variability, quality and punctuality needs of the modern customer, production requirements make it evident that traditional automation solutions can't solve all challenges [1].

Machines nowadays operate in *complex* and *dynamic* environments, *quickly* producing a wide *variety* of *quality* products for *demanding* customers. As *these* requirements continue to increase, it becomes impossible to remain scalable and sustainable using traditional automation solutions [2].

Instead, modern automation solutions should utilize efficient planning algorithms that compute schedules and sequences of operations automatically. Planning is a deliberative decision making process which yields sequences of operations that drive state change towards a goal.

Automated planning is already implemented in some modern solutions, however the idea of planning isn't new since algorithms that compute plans based on explicit state-space searches exist since the late '50s [3], and symbolic methods based on BDDs since the late '70s [4].

A more recent method that has established itself as an important planning approach is SAT-based planning. Planning problems are encoded as satisfiability problems and the results are calculated by SAT solvers.

Even though SAT-based planning was first proposed by Kautz and Selman already in 1992 [5], the interest of planning researchers in SAT-based planning methods was limited up until recently. One of the main reasons behind this was the performance advantage of explicit state-space search over solving early SAT encodings of planning problems [6]. However, modern planners based on satisfiability match, and often outperform, planners based on other search paradigms [7].

Explicit state-space search and symbolic methods based on BDDs are known for their performance in solving problems with a small number of state variables, however SAT-based methods excel in solving hard combinatorial planning problems with a relatively high numbers of state variables [8]. Another advantage of planners based on SAT is that algorithms used to compute plans are almost completely general purpose SAT solving algorithms, which means that every improvement in the solver directly improves planning.

However, a known issue with SAT-based automated planning is that plan calculation seems to slow down significantly as the plan length increases [6]. In an effort to avoid this limitation while utilizing the strengths of SAT-based planning, we present a compositional algorithm that divides a planning problem into a number of simpler problems that are faster to solve. We do this with a combination of abstraction refinement using activation parameters and step-wise problem generation, resolution and concatenation. We test the proposed algorithm on a number of examples and show that in a lot of cases a significant speed-up [9] is achieved.

In the following section, we present a version of an existing incremental planning algorithm that we use as the low-level solving engine in our approach. In Section 3, the high-level compositional algorithm is introduced. In the section after that, we test our approach on a number of examples and compare planning performances of the compositional algorithm against the algorithm of Section 2. We discuss certain benefits, drawbacks and possible future improvements of the compositional algorithm in section 5. There, we also mention relevant publications in the area of compositional planning and planning with abstraction refinement. Finally, we conclude the paper in Section 6.

II. Incremental Planning

We utilize a simple incremental planning algorithm based on [10] to solve individual problems generated by the compositional algorithm. The incremental algorithm tries to find a plan by testing the satisfiability of the planning problem's formulas for a sequentially increasing horizon length. It utilizes an incremental SAT-solver that makes it possible to add a new time point in each step so that the SAT-solver can learn from previous attempts.

*This work has been supported by Unification and Unicorn, VINNOVA.

[1]Endre Erős, Martin Dahl, Petter Falkman and Kristofer Bengtsson are with Chalmers University of Technology, Electrical Engineering, Systems and Control Department, Automation Research Group, Gothenburg, Sweden. *endree@chalmers.se*, *martin.dahl@chalmers.se*, *petter.falkman@chalmers.se*, *kristofer.bengtsson@chalmers.se*

The version presented in this paper also lets the algorithm create backtracking points that enables it to manipulate the content of the context between each step, i.e. to remove certain clauses that are added after a point by backtracking to that point. For example, if a planning attempt fails, this feature is used to remove goal clauses for the current step before adding new goal clauses for the next step. Before describing the incremental algorithm, it is necessary to define a few basic concepts. Let's do this with an example.

Example: A robotic manipulator transports products from a buffer to a fixture where the products are processed. In order to control the robot, we have to read its current *pose* as well as to send *pose* commands to it. To establish this two way communication, a *pose command* and a *pose measured* variable is used. Let's call these variables $pose_c$ and $pose_m$.

The real sensor-level measurement from the robot is discretized into three discrete values that can be assigned to the variable $pose_m$. The robot can be at the *buffer (b)*, at the *fixture (f)* or at *unknown (u)* if it is somewhere between the *buffer* and the *fixture*.

Definition 2.1: A *variable* v_i is a named unit of data that can be *assigned* a value from its finite domain of values v_i^D.

Example: In our robot scenario, the values *buffer*, *fixture*, and *unknown* constitute the finite *domain* of the variable $pose_m$. We don't want to command the robot to go to the *unknown* pose, so only the poses *buffer* and *fixture* make up the domain of the variable $pose_c$.

Definition 2.2: A *complete variable set* V_c for a system is a set of all variables defined for that system. A *partial variable set* V_p is a non-empty subset of V_c.

Definition 2.3: The *complete state space* V_c^D of a system is the Cartesian product of all value domains of variables in V^c defined for that system. A *partial state space* V_p^D is a non-empty subset of V_c^D.

Example: With the currently defined variables in our robot system, the *complete variable set* and the *complete state space* of our system is:

$$V_c = \{pose_m, pose_c\}$$
$$V_c^D = \{\langle b,b\rangle, \langle f,b\rangle, \langle u,b\rangle, \langle b,f\rangle, \langle f,f\rangle, \langle u,f\rangle\}$$

Now that we have defined the states our system can be in, it is time to model how the system can move between those states. Before we can do that, we have to define *predicates*, which are the fundamental building blocks of our *models*.

Definition 2.4: A *predicate* is a logical expression of one or more variables. A predicate evaluates to *true* if the assignments of its variables satisfy the logical expression, and to *false* otherwise.

Example: Let's build a few simple predicates and name them so that we can reuse them later. Something we might reuse a lot is knowing whether the robot is at the buffer, fixture or somewhere between:

$$r\_at\_b := pose_m == b$$
$$r\_at\_f := pose_m == f$$
$$r\_at\_u := pose_m == u$$

These predicates can either be *true* or *false*, depending on the current value of the variable $pose_m$ during evaluation. We can also build more complex predicates and name them as we like. For example, another thing we might find useful during modeling is knowing whether the robot has been issued a command to move to the *buffer*. With the current *complete variable set* that we have defined for this system, we can say that the robot is moving towards the buffer if:

$$r\_moving\_to\_b := \neg r\_at\_b \land pose_c == b \tag{1}$$

Before we define *transitions*, let's look at how a very simple plan would look like. As it was said before, a *plan* is a sequence of transitions driving state change towards a goal. In this example, the robot moves from the *buffer* to the *fixture*.

$$state_0 : pose_m == b \land pose_c == b$$
$$trans_1 : start\_move\_robot\_to\_fixture$$
$$state_1 : pose_m == b \land pose_c == f$$
$$trans_2 : finish\_move\_robot\_to\_fixture$$
$$state_2 : pose_m == f \land pose_c == f$$

This sequence has several steps with each step being one transition leading to a state. The chain of states is called a *trace* and since only one transition is allowed to be taken at a time, the states are *temporally* related. This means that we can say that $state_2$ is the *next* state after $state_1$.

Definition 2.5: A *transition* t is a predicate:

$$t = g \land e \tag{2}$$

where g is a *guard* predicate and e is an *effect* predicate. If the transition is to be *taken*, the *guard* predicate has to evaluate to *true* for the *current* step, while at the same time, the *effect* predicate has to evaluate to *true* for the *next* step.

Example: Let's look at the *start_move_robot_to_fixture* transition. This transition is modeled as:

$$start\_move\_robot\_to\_fixture =$$
$$pose_{m_i} == b \land pose_{c_i} == b \land pose_{c_{i+1}} == f$$

where the *guard* predicate is marked with i for the current step and the *effect* predicate with $i+1$ for the next step. We finally have the necessary components to define a planning problem.

Algorithm 1: ***Incremental***

```
Input: (i, g, T, $s_{max}$)
Output: planning result
let step := 0;
let ctx := create context;
add constraint (ctx, i, step);
let bp := create backtracking point;
add constraint (ctx, g, step);
while step $\leq s_{max}$ do
    step += 1;
    if check(ctx) == UNSAT then
        backtrack to level bp;
        let t_disj := disjunction for T;
        add constraint (ctx, t_disj, step);
        let bp := create backtracking point;
        add constraint (ctx, g, step);
    else
        return planning result;
        break;
    end
end
return empty planning result;
```

Definition 2.6: A *transition system* T for a given system is a collection of all transition predicates that model the behavior of that system.

Definition 2.7: A *planning problem* Ψ is a 4-tuple:

$$\Psi = \langle i, g, T, s_{max} \rangle \tag{3}$$

where i and g are initial and goal predicates, T is the transition system, and s_{max} is a limit on the horizon length. Actually, Ψ is a 5-tuple containing S as well, where S is a collection of specifications modeled as LTL_f formulas encoded in SAT [11]. However, specifications are omitted in this paper for the sake of brevity, so we will refer to a planning problem as it is defined in (3).

The incremental algorithm takes a planning problem Ψ and either returns a complete result of the planning problem, or an empty result which represents that no solution was found.

An integer variable *step* keeps track of the step in the plan that the algorithm is currently at. At line 2 of Algorithm 1, a context *ctx* is created for the problem so that the solver can keep track of assertions.

As you can see at lines 3 and 5, the algorithm *asserts* the initial and goal constraints for *step-0* into the context. It also creates a backtracking point in line 4 before asserting the goal, so that it can be removed from the context if a solution is not found in the current step.

Inside a loop that ensures the horizon limit is not exceeded, the algorithm increments the *step* and *checks* if the current assertions in the context are consistent. If the assertions are SAT in the first step, that means that the variable assignments satisfy both the goal and the initial predicates.

Otherwise, the assignments in the goal predicate for *step-0* are not consistent with the assignments in the initial predicate, so the goal is for the current step is removed them from the context by backtracking to the previous point.

The solver checks the transitions in a step from the disjunction of all transitions in the model. If any transition in the disjunction satisfies the current assignment, *planning* goes on. Semantically, it can be said that the transition is *taken*.

Only one transition is allowed to be taken in each step from the disjunction of all transitions in the model. Practically, transitions are tracked in each step with Boolean-valued variables, so that by the time a plan is found, we know which transition was evaluated to *true* in which step. This is done by conjuncting the transition with a Boolean-valued variable, so if the transition is *taken* in a step, that variable has to be true in that step.

The algorithm doesn't know if the next goal assignment will be consistent with the assignments that are currently in the context, so it creates a new backtracking point before it assigns the goal for *step-1* into the context.

As you can see, this process is repeated while the horizon length sequentially increases. If a solution for the problem is found in a *step* that is less than the horizon length limit s_{max}, it is returned by the algorithm. Otherwise, the limit is breached and an empty result is returned.

It is important to limit the planning horizon so that the algorithm can terminate in case a solution can't be found, or where it takes a long time to calculate it.

III. Compositional Planning

As mentioned before, the main idea behind the compositional algorithm shown as Algorithm 2 is to break the planning problem into simpler problems that can be solved fast. To solve these individual simple planning problems, the incremental algorithm from the previous section is used.

Figure 1 serves as an example and a visual guide to explain how the compositional algorithm works and what is happening in different steps. To keep track of these steps, we refer to the alphabetic annotations on the right side of the same figure.

A. Organization

In order for the compositional algorithm to refine, generate and solve parts of the complete planning problem, we have to allow its abstraction and refinement. The planning problem (3) is *parameterized* so that the compositional algorithm can enable or disable certain *basic predicates* in order to generate abstracted input constraints to the incremental algorithm. In order to do this, a number of *partial variable sets* is defined during modeling.

Definition 3.1: A *basic predicate* is a predicate of variables from only one partial variable set.

Defining partial variable sets depends on some expert knowledge of the planning problem, hence it is a part of the modeling process. For example, we choose to group variables together into partial variable sets so that we can form basic

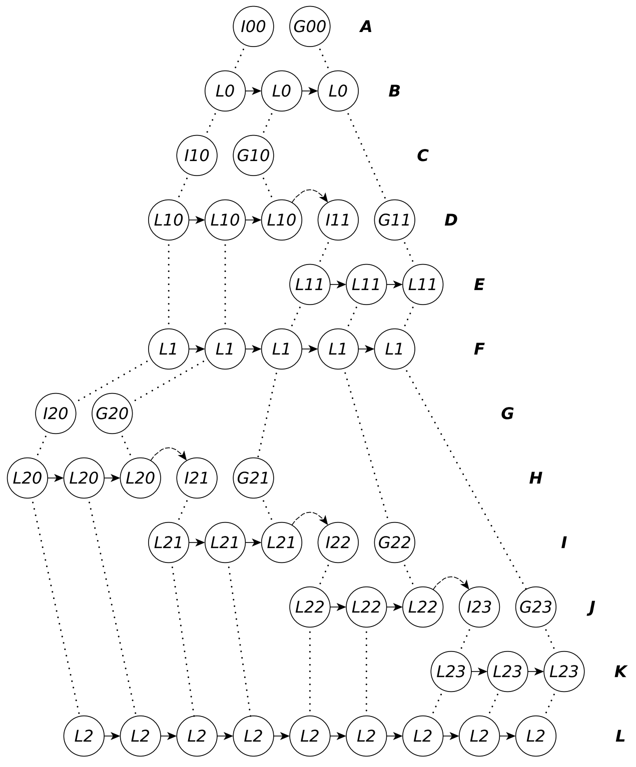

Fig. 1. How the compositional algorithm works

predicates that contain none other but variables from the same set. This allows us to compose more complex *parameterized predicates* and at the same time keep track of which partial variable sets play a role in them.

Example: Let's extend our example by adding more functionality to our system. For example, as a safety feature, let's add the option to control the robot's *status* which can enable or disable its movement. For this, we define two new variables, $stat_c$ and $stat_m$ with the same value domain $\{enabled, disabled\}$.

Moreover, let's equip the robot with a sensor in order to know whether it is holding a product or not. In fact, let's equip also the buffer and the fixture with the same type of sensors so that we can always track where the products are. For this, we define three additional variables, $grip_m$, $buff_m$ and $fixt_m$ with the same value domain $\{empty, full\}$.

In this example, grouping the variables into partial variable sets comes naturally. Let's name these sets to make it easier to reuse them later:

$$\begin{aligned} pose &= \{pose_c, pose_m\} \\ stat &= \{stat_c, stat_m\} \\ prod &= \{grip_m, buff_m, fixt_m\} \end{aligned}$$

A good example of a *basic predicate* would be (1) since it only contains variables from the $pose$ partial variable set. In order to know which variables are in a basic predicate, the name of the partial variable set they belong to will be present as a superscript in the names of basic predicates:

$$r\_moving\_to\_b^{pose} := \neg r\_at\_b \wedge pose_c == b$$

We have defined the building blocks that enable us to generate abstracted planning problems for the incremental algorithm.

B. Parameterization

In order to enable or disable certain *basic* predicates, we define *activation parameters* that enable or disable parts of *parameterized predicates*.

Definition 3.2: An *activation parameter* is a Boolean-valued variable that is used to hide (*false*) or reveal (*true*) basic predicates in a parameterized predicate.

Definition 3.3: A *parameterized predicate* is a set of 2-tuples:

$$\{\langle bp_1, a_1\rangle, \langle bp_2, a_2\rangle, ..., \langle bp_n, a_n\rangle\} \tag{4}$$

where $bp_1, bp_2, ..., bp_n$ are basic predicates and $a_1, a_2, ..., a_n$ are their respective activation parameters.

One activation parameter is usually defined for each partial variable set. Hence, an alternative notation will be used for parameterized predicates throughout the rest of the paper:

$$\{bp_1^{a_1}, bp_2^{a_2}, ..., bp_n^{a_n}\} \tag{5}$$

Definition 3.4: A *parameterized transition* t_P is a transition whose guard and effect are parameterized predicates.

Example: Let's build a parameterized transition that models how the robot takes products from the buffer.

$$\begin{aligned} &take\_product\_from\_buffer = \\ &\quad \{b\_full_i^{prod}, r\_empty_i^{prod}, active_i^{stat}, activate_i^{stat}, \\ &\quad\; r\_at\_b_i^{pose}, r\_go\_to\_b_i^{pose}, r\_full_{i+1}^{prod}\} \end{aligned}$$

In order for the robot to hold a product in the next step ($r\_full_{i+1}^{prod}$), several things have to be fulfilled in the current step. The buffer must hold a product ($b\_full_i^{prod}$) that the empty robot can take ($r\_empty_i^{prod}$). In order to do anything, the robot has to be active ($active_i^{stat}$) and in order to take a product from the buffer, it has to be at the buffer ($r\_at\_b_i^{pose}$).

Meanwhile, we don't want to cause changes in some variables in the next state. The basic predicate ($activate_i^{stat} := stat_{c_i} == enabled$) ensures that the measured variable $stat_m$ will hold its $enabled$ value in the next step. The same goes for the $pose_m$ variable.

Definition 3.5: A *parameterized transition system* T_P is a transition system whose transitions are parameterized transitions.

Definition 3.6: A *parameterized planning problem* Ψ_p is defined as:

$$\Psi_P = \langle i_P, g_P, T_P, P, s_{max} \rangle \tag{6}$$

where i_P and g_P are initial and goal parameterized predicates, T_P is a parameterized transition system, P is a list of activation parameters and s_{max} is the limit on the plan length that is applied to every generated problem sent to the incremental algorithm.

Now that we have modeled a parameterized planning problem, we let activation parameters enable or disable basic predicates in parameterized predicates, generating abstracted input constraints for the incremental algorithm by conjuncting these enabled basic predicates.

C. Activation

The compositional algorithm takes a list of activation parameters P, activates the next parameter in the list, generates and solves problems. Hence, the order of the activation parameters in the list P decides the problem refinement order.

Example: Let's take the parameter list:

$$P = (prod, stat, pose)$$

and $take\_product\_from\_buffer$, the parameterized transition that we have built earlier. This transition contains seven basic predicates, where the first six are constituting the guard, and the last one the effect. While generating the real input predicate for the incremental algorithm as a conjunction of these basic predicates, the values of their respective activation parameters determine whether the basic predicate is included in the conjunction or not.

The activation parameter list P has three parameters that are initially disabled, hence the *Activate* procedure from Line 3 of Algorithm 2 activates the first parameter $prod$, enabling the basic predicates $b\_full_i^{prod}$, $r\_empty_i^{prod}$ and $r\_full_{i+1}^{prod}$ to take part in the generated conjunction that is the input transition for the incremental algorithm.

We can refer to this step-wise parameter activation as *refinement*. Each step of the compositional algorithm is called a *level*, and in each level, the problems that are generated and sent to the incremental algorithm are more refined, meaning that more variables play a role in the generated predicates of a problem. For instance, let's follow the complete refinement of the generated transition:

$$b\_full_i^{prod} \wedge r\_empty_i^{prod} \wedge r\_full_{i+1}^{prod}$$

In the next level, the algorithm activates the $stat$ parameter, so the generated transition is more refined:

$$b\_full_i^{prod} \wedge r\_empty_i^{prod} \wedge$$
$$active_i^{stat} \wedge activate_i^{stat} \wedge r\_full_{i+1}^{prod}$$

Finally, in the last level, the algorithm activates the $pose$ parameter and the generated transition is completely refined. The completely refined transition is a conjunction of all predicates from its parameterized counterpart.

Algorithm 2: ***Compositional***

```
Input: ($i_P$, $g_P$, $T_P$, P, $s_{max}$)
Output: planning result
1  let level := 0;
2  let (i, g, T, P) := Activate($i_P$, $g_P$, $T_P$, P);
3  let r := Incremental(i, g, T, $s_{max}$);
4  Plan(r, $i_P$, $g_P$, $T_P$, P, $s_{max}$, level);
5  function Plan(r, $i_P$, $g_P$, $T_P$, P, $s_{max}$, level) begin
6  if not all parameters activated then
7      if r.plan_found then
8          let h := := new empty list;
9          let concat := 0;
10         let level_results := new empty list;
11         (i, g, m, P) := Activate($i_P$, $g_P$, $T_P$, P);
12         for j in (0 to (r.trace.length - 1)) do
13             let $g_l$ := r.trace(j + 1);
14             concat += 1;
15             if j == 0 then
16                 let $r_P$ := Incremental(i, $g_l$, T, $s_{max}$);
17                 let h := $r_P$.trace.tail;
18                 level_results.push($r_P$);
19             else if j == r.trace.length - 1 then
20                 let $r_P$ := Incremental(h, g, T, $s_{max}$);
21                 level_results.push($r_P$);
22             else
23                 let $r_P$ := Incremental(h, $g_l$, T, $s_{max}$);
24                 let h := $r_P$.trace.tail;
25                 level_results.push($r_P$);
26             end
27         end
28         let $r_{loopy}$ := Concatenate(level_results);
29         let $r_{filt}$ := Filter($r_{loopy}$);
30         Plan($r_{filt}$, $i_P$, $g_P$, $T_P$, P, $s_{max}$, level + 1);
31     else
32         return empty planning result;
33     end
34 else
35     return r;
36 end
```

D. Generation

As you can see from line 4 of Algorithm 2, the *Plan* function receives the planning result of the previous level, and for each step in the calculated plan it generates a new planning problem. This generation doesn't occur at once, since the following problem depends on the result of the previous problem in the same level. We could say that the following problem *inherits* information from the previous result.

Three cases are differentiated while generating problems depending on the place of the step in the solution of the previous level. We refer to these cases as the *first*, *central* and *last* problems of a level. A *first* case problem is generated from the first step in the plan of the previous level. For example, this can be seen at point *C* in Figure 1. Similarly, a *last* case problem comes from the last step in the plan of the previous level as it can be seen at point *D*. All other problems that are generated from steps between the first and the last step are *central*. This can be seen at points *H* and *I* in Figure 1.

Steps in the plan hold transitions that change the state towards a goal. Sample plans at certain levels of the compositional algorithm can be seen at lines *B*, *F* and *L* in Figure 1. Each transition in a plan has a *source* and *sink* state. The three different cases of problems in a level come from the way the *initial* and *goal* predicates of a problem are generated from these *source* and *sink* states of a transition in a step.

1) First case problem: When the algorithm is generating a *first* case planning problem, the initial predicate i of this problem is generated from the initial parameterized predicate i_P after the next parameter has been activated. In essence, it is a refined version of the initial predicate from the previous level. This is happening at line 16 of Algorithm 2.

The goal predicate of the *first* case planning problem is generated from the *sink* state of the transition in the first step, and it is a conjunction of assignments that make up that state. You should note that this new *goal* predicate doesn't contain variables that play a role in the initial predicate of the same problem. In order to concatenate results after they are calculated, the goal predicate of the *first* case planning problem is not provided with assignments for the variables that are being included in this level.

If we would to provide assignments for these variables in the goal predicate of the *first* case problem, there would probably be a discrepancy between assignments of adjacent plans of the level during concatenation. We would have to guess the assignments for those variables and in the end, we would have to plan between plans to achieve a correctly concatenated solution. That is why we leave it to the solver to decide an assignment to the new goal predicate variables included in the next level that is consistent with the other assignments.

2) Last case problem: When the algorithm uses the last step in the plan of the previous level to generate a problem, that is a *last* case planning problem. This is happening at line 20 of Algorithm 2 and at points *D* and *J* in Figure 1. Since no further problems will follow in this level, the goal predicate g of this problem is generated from the goal parameterized predicate g_P after the next parameter has been activated. In essence, it is a refined version of the goal predicate from the previous level.

The initial predicate of the *last* case planning problem is generated from the *source* state of the transition in the last step, and it is a conjunction of assignments that make up that state. This time, we are providing the *initial* predicate with additional assignments by using an *inheritance* variable h to pass assignments between problems and solutions of a level, as you can see in lines 18 and 25 of Algorithm 2. At this point, after the solver has found satisfiable assignments for the goal state of the previous problem of the same level, the new assignments are *inherited* by the next planning problem to play a role the *initial* predicate, as you can see at points *D, H, I* and *J* in Figure 1.

3) Central case problem: When the algorithm generates a *central* case planning problem, the initial predicate is generated the same way as the initial predicate in the *last* case problem, and the goal predicate the same way as the goal predicate in the *first* case problem.

E. Resolution

To solve each generated problem of a level, we use the Algorithm 1 from the previous section. In lines 4, 17, 21 and 24 of Algorithm 2, this is indicated with *Incremental*. You can see in lines 16, 20 and 23 of Algorithm 2 how the algorithm distinguishes problem cases based on where the step of the plan of the previous level is located. For example, *first* case problems are generated at points *C* and *G* and solved at points *D* and *H* in Figure 1. Similarly, *last* problems are generated at points *D* and *J* and solved at points *E* and *K*, and *central* problems are generated at points *H* and *I* and solved at points *I* and *J* in Figure 1. The important thing to notice is that in each level, problem generation and resolution happen iteratively one after the other until all solutions have been found for that level, or until one of the resolutions fail. This iterative generation, inheritance and resolution can best be seen between points *G* and *K* in Figure 1.

F. Concatenation

In each level, new planning problems are generated from the result of the previous level. As these problems are solved, we end up with a number of results that we have to concatenate in order to get the complete result of that level. This concatenation of plans represents the complete plan of the current level, and it is indicated with *Concatenate* in line 29 of Algorithm 2, which is happening at points *F* and *L* in Figure 1.

To keep track of where results should fit in the complete result of a level, we use a *concat* variable, as you can see in line 15 of Algorithm 2. After all problems of a level are solved, we concatenate the individual plans in the right order using the *concat* variables to get the plan of the level.

G. Filtering

In some cases, the result after concatenation might contain certain sections that are redundant. These sections lead to a duplicate of a state that was already reached earlier in the trace, so you can look at these sections as *loops*.

If loops appear in the concatenated result, the algorithm filters them out of the plan using the *Filter* procedure as you can see in line 30 of Algorithm 2. These *loops* appear sometimes after concatenation as a result of solving individual problems of a level while satisfying all specifications at that level. In planning, it is not of our interest to visit a logical state more than once, so if a loop appears in the concatenated trace, the algorithm removes it. After we filter out the loops, the plan is still consistent with all specifications at that level.

Once all parameters are activated, the problem is completely refined. After the last check of the parameter list as seen in line 6 of Algorithm 2, if a plan is found in the previous level, it is returned by the algorithm in line 35.

IV. Evaluation

Several planning problems are solved using both the compositional and the incremental algorithms. In this section, we are evaluating some of the results and comparing the length and quality of plans calculated by the incremental and the compositional algorithm.

A. Example 1

Let's test the algorithms on the robot example used in this paper. For brevity, the names of partial variable sets are shortened to one letter: *s* for *stat*, *i* for *prod* and *p* for *pose*.

For all the variations in the refinement order, the compositional algorithm calculates the same 18 step long plan as the incremental algorithm. The different benchmarks are derived from at least 10 runs for each test and the name of the test suggests the refinement order in the example. You can see from the benchmarks that the refinement order influences the plan calculation time, which is expected.

```
Benchmark:           Time (mean ± dev):
test_1_inc           316.8 ms ± 14.3 ms
test_1_comp_isp      175.9 ms ± 8.4 ms
test_1_comp_ips      268.0 ms ± 14.1 ms
test_1_comp_sip      174.1 ms ± 11.0 ms
test_1_comp_spi      234.5 ms ± 15.2 ms
test_1_comp_psi      274.8 ms ± 4.3 ms
test_1_comp_pis      351.8 ms ± 7.7 ms
```

B. Example 2

Let's extend Example 1 with several additional complications. Now, we are able to control and measure the pose of the gripper, as well as to control and measure its status. Additionally, in order to avoid collision, the robot has to move through three *via* points. In this example, we have five partial variable sets: *s* for *stat*, *i* for *prod*, *p* for *pose*, *g* for *grip_pose* and *m* for *grip_stat*.

Provided is a short list of results that yielded the same 38 step long plan while testing this example. You can see from the benchmarks that as much as a 20x speed-up can be achieved with a good refinement order.

```
Benchmark:           Time (mean ± dev):
test_2_inc           13.461 s ± 0.500 s
test_2_comp_mgsip    1.026 s  ± 0.011 s
test_2_comp_gmpsi    908.4 ms ± 7.2 ms
test_2_comp_gmisp    671.3 ms ± 9.8 ms
```

V. Discussion

We found several publications that share similar opinions with us to be, to the best of our knowledge, state-of-the art results in areas of compositional planning and planning with abstraction refinement.

The authors of [12] use a counterexample guided abstraction refinement method to solve planning problem instances encoded in SAT. They obtain a relaxed instance by removing clauses from the model that is in conjunctive normal form and refine it later using counterexamples found during the search.

In [13], the authors distinguish between planning problems with and without symmetries. They decompose a planning problem with symmetries into a set of abstracted, isomorphic subproblems. After solving each abstraction, they concatenate the results together to yield a solution for the original problem.

This paper is our first contribution in a series that tries to tackle the problem of safe and non-blocking online planning, and as such, it only focuses on computing a plan faster. Safety and non-blocking properties are well known problems, however they will be addressed in future papers in order to limit the scope of this work.

What we show is that there are cases when the compositional algorithm clearly outperforms the well known incremental algorithm in terms of computation time. However, there are a few known shortcomings which we will mention now.

A. Refinement order matters

As you can see from the benchmarks in Examples 1 and 2, defining a good refinement order can influence the planning time quite much. Solving the same problem with a different refinement order gives quite different plans before filtering out the loops.

Usually, after filtering out the loops, the plans calculated after these two refinement orders are the same. However, it happens sometimes that the refinement order influences the final plan length as well. In these cases, the yielded plan is still correct, however it is not of the minimal length.

B. No optimality guarantee

If there are several valid plans of different lengths that can be calculated at a certain step, a shortest one will be yielded. However, this doesn't mean that this will yield the shortest plan of a level. In some cases, the compositional algorithm can provide a plan in a level that is short, however after refinement it would turn out that the solution after the last refinement is longer than the result the incremental algorithm would yield. This happens because of the breadth-first search nature of the incremental algorithm.

VI. Conclusion

This paper presents a high-level compositional implementation that utilizes the an incremental SAT-based planning algorithm as its solving engine to perform automated planning. This is achieved by generating abstracted problems from the main parameterized planning problem, solving them using the incremental solver and concatenating the results in order to achieve a complete plan. We show that in a majority of cases, a significant speed-up is achieved.

Primarily, our plan for the future is to research the influence of the refinement order in this approach. We would like to develop a rule about defining a good refinement order, since solving a problem for all refinement order variations is not feasible, even for a small number of activation parameters.

Secondly, we will investigate safety and non-blocking properties of compositional planning and lastly, we would like to test the algorithm on standard planning benchmarks.

References

[1] A. Hanna, K. Bengtsson, M. Dahl, E. Erős, P. Götvall, and M. Ekström, "Industrial challenges when planning and preparing collaborative and intelligent automation systems for final assembly stations," in *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2019, pp. 400–406.

[2] A. Azizi, *Applications of Artificial Intelligence Techniques in Industry 4.0*, 1st ed. Springer Publishing Company, Incorporated, 2018.

[3] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, no. 1, p. 269–271, Dec. 1959. [Online]. Available: https://doi.org/10.1007/BF01386390

[4] S. Akers, "Binary decision diagrams," *Computers, IEEE Transactions on*, vol. C-27, pp. 509 – 516, 07 1978.

[5] H. Kautz and B. Selman, "Planning as satisfiability," in *Proceedings of the 10th European Conference on Artificial Intelligence*, ser. ECAI '92. USA: John Wiley & Sons, Inc., 1992, p. 359–363.

[6] J. Rintanen, "Planning as satisfiability: Heuristics," *Artificial Intelligence*, vol. 193, pp. 45 – 86, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370212001014

[7] ——, "Madagascar: Scalable planning with sat," *Proceedings of the 8th International Planning Competition (IPC-2014)*, vol. 21, 2014.

[8] ——, "Search methods for classical and temporal planning," *Tutorials of the 21th European Conference on Artificial Intelligence (ECAI 2014)*, vol. 21, 2014.

[9] ——, "Evaluation strategies for planning as satisfiability," in *Proceedings of the 16th European Conference on Artificial Intelligence*, ser. ECAI'04. NLD: IOS Press, 2004, p. 682–686.

[10] S. Gocht and T. Balyo, "Accelerating sat based planning with incremental sat solving," in *ICAPS*, 2017.

[11] J. Li, K. Rozier, G. Pu, Y. Zhang, and M. Vardi, "Sat-based explicit ltlf satisfiability checking," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 2946–2953, 07 2019.

[12] N. Froleyks, T. Balyo, and D. Schreiber, "Pasar - planning as satisfiability with abstraction refinement," in *Proceedings of the 12th Annual Symposium on Combinatorial Search (SoCs 2019), Napa, CA, July 16-17, 2019*. AAAI Press, Menlo Park, CA, 2019, pp. 70–78.

[13] M. Abdulaziz, C. Gretton, and M. Norrish, "A verified compositional algorithm for ai planning," in *ITP*, 2019.