Fully Automated Cyclic Planning for Large-Scale Manufacturing Domains

Masataro Asai and **Alex Fukunaga**
Department of General Systems Studies
Graduate School of Arts and Sciences
The University of Tokyo

Abstract

In domains such as factory assembly, it is necessary to assemble many identical instances of a particular product. While modern planners can generate assembly plans for single instances of a complex product, generating plans to manufacture many instances of a product is beyond the capabilities of standard planners. We propose ACP, a system which, given a model of a single instance of a product, automatically reformulates and solves the problem as a cyclic planning problem. We show that our domain-independent ACP system can successfully generate cyclic plans for problems which are too large to be solved directly using standard planners.

1 Introduction

Domain-independent planning is a promising technology for assembly planning in complex, modern robotic cell-assembly systems consisting of multiple robot arms and specialized devices that cooperate to assemble products [Ochi et al., 2013]. In a small-scale, feasibility study, Ochi et al showed that although standard domain-independent planners were capable of generating plans for assembling a single instance of a complex product, generating plans for assembling multiple instances of a product was quite challenging. For example, generating plans to assemble 4-6 instances of a relatively simple product in a 2-arm cell assembly system pushed the limits of state-of-the-art domain-independent planners. However, real-world cell-assembly applications require mass production of hundreds/thousands of instances of a product.

Ochi et al proposed a (cyclic) "steady-state" (SS) model, where the problem of generating a plan to manufacture many instances of a product is reformulated as a cyclic planning problem. In an instance of the general cyclic planning/scheduling problem [Draper et al., 1999], the start and end states of the planning instance correspond to a "step forward" in an assembly line (See Fig. 1), where partial products start at some location/machine, and at the end of this "step", (1) all of the partial products have advanced forward in the assembly line (2) one completed product exits the line, and (3) assembly of a new, partial product has begun. Ochi et al showed that when an appropriate, *manually generated*, start/end state for this cyclic planning instance was provided to a planner, the resulting manufacturing process was competitive with a human-generated, cyclic assembly plan. However, identification of the "steady-state", the crucial component of this approach, was an entirely manual process – the planner was only responsible for computing paths between the cycle start/end points, so the overall process was far from automated.

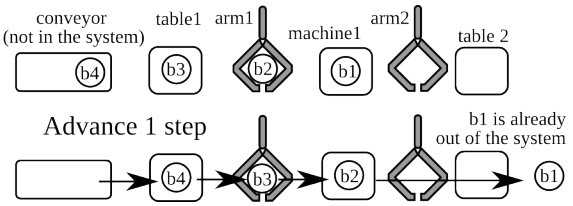

Figure 1: An example of a start/goal of a steady state with four products b1,b2,b3 and b4

This paper proposes ACP (Automated Cyclic Planner), which fully automates the cyclic scheduling process for "mass manufacturing", e.g., cell assembly. Starting with a standard PDDL model for a single instance of a product, ACP analyzes the model, automatically extracting the structure necessary to identifying all possible steady-states that can be used in a cyclic plan. ACP then filters the set of candidate states and searches for one which can be used in an efficient, parallel plan for manufacturing an arbitrary number of instances of the product. Although ACP is motivated by the cell assembly domain, ACP is a domain-independent system which can be applied to other domains which require mass assembly of a single product. We first briefly describes the cell assembly domain. We then describe the domain analysis implemented in ACP, focusing on the formal criteria used to identify candidate steady-states, as well as the methods used to filter them. We experimentally evaluate ACP on the cell assembly domain, a modified version of the woodworking IPC'11 domain, and unmodified IPC barman domain. We show that ACP enables fully automated generation of large-scale assembly plans for problems which are totally beyond the reach of standard, state-of-the-art planners.

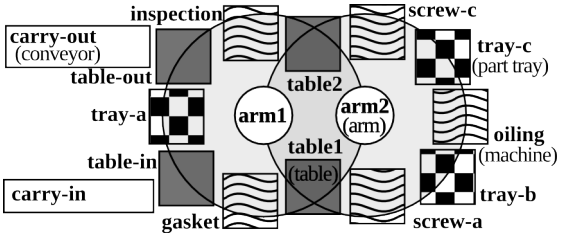

Figure 2: Example CELL-ASSEMBLY instance: model2a.

2 CELL-ASSEMBLY Domain

CELL-ASSEMBLY is a PDDL domain with STRIPS-style actions, negative-preconditions, action costs, and a hierarchical type structure[McDermott et al., 1998].

Tasks in the CELL-ASSEMBLY domain consist of assembling many products, called "bases", using several robot arms and machines. Each base is first pushed into the system by a "carry-in" conveyor. Components may be attached to the base on the "tables" and the finished product is carried out of the system by the "carry-out" conveyor. The bases are always held by an arm or "set" on a table or a machine. Each table/machine can hold only 1 base at a time, and each arm can hold only 1 object at a time.

During assembly, bases are processed by a set of machines, each with a specific purpose such as painting, oiling, inspection etc. Thus various kinds of steps are required to complete a product, which must be performed in a specific order described by "job" dependencies. An example of a CELL-ASSEMBLY plant is shown in Fig. 2.

The range of motion of each arm is limited as shown by the large circles around the arms in Fig. 2. The job dependencies are statically stated in a initial state of a problem with (depends ?job ?prev-job) predicates. The predicate (finished ?job ?base) means one such manipulation was done on the base. Each base is initialized to have a fact (finished nothing-done base) in any problems. None of bases exit the system until (finished j_n base) is true for all jobs j_n. For example, when there are 5 jobs from j_0 to j_4 and a base b, then the goal condition contains (finished j_n b) for $0 \leq n \leq 4$ and (finished nothing-done b).

There are 6 actions in this domain: 4 operators for physical movement, (move-arm ?arm ?from ?to), (set-base ?base ?arm ?pos), (eject-base ?base ?arm ?pos), (pickup-component ?component ?arm ?pos), as well as 2 sequencing operators assemble-with-machine and assemble-with-arm.

Since the job dependencies (encoded as preconditions on sequencing operators) specify the ordering of the assembly process, planning the efficient movement of the robot arms that move bases/parts through the assembly process is the primary task left to the planner [Ochi et al., 2013].

3 Overview of ACP

ACP takes as input a *manufacturing order*, which consists of a PDDL domain, a name of a single instance of a *product*, a typed PDDL problem file specifying a *model* for it, and N, the number of instances of the product to manufacture. Currently, we support STRIPS-style actions, negative preconditions and action costs. The output of ACP is a plan for manufacturing N instances of the product.

ACP currently assumes the following:

- *Single Product Type per Order:* As an input, ACP takes the order to assemble N instances of a particular product, e.g., "assemble 10 units of widget A". It does not handle the mixed orders in which multiple types of products are assembled simultaneously, e.g., "assemble 10 units of widget A and 7 units of widget B".
- *Uniform Manufacturing Process:* To fulfill the order, all instances of product must be assembled in the exact same order using the exact same machines. Suppose that there are 2 possible plans to assemble a product: (1) attach part p_1 first then p_2; (2) attach part p_2 first, then p_1. Another possible kind of variation may be (1) do job j at machine m_1 ; (2) do the same job j at m_2 . ACP chooses one of those possibilities first and always applies the same plan (and it never mixes multiple assembly plans.)
- *Indistinguishable Parts:* Suppose a widget can be assembled from a base and two parts, part1 and part2. ACP assumes that *instances* of all components are indistinguishable, i.e., all the bases are identical to each other, and all instances of part1 are identical to all other instances of part1. In other words, while we can specify "attach some instance of part1 to each base", we cannot specify: "attach this particular instance of part1 to this particular base".

At a high level, ACP performs the following steps:

1. A standard domain-independent planner is used to find a plan for manufacturing a single instance of the product. This is the *template plan* which is used as the basis for the cyclic plan.
2. ACP analyzes the template plan using the name of the product as well as the original input PDDL. It extracts the structures that are necessary in order to specify start/end points for cyclic plans.
3. Based on the analysis in the previous step, a set of candidate steady-state start/end points for cyclic planning are constructed. This large set of candidates are pruned to a manageable number using some filtering heuristics.
4. Each remaining candidate steady-state is evaluated by solving a temporal problem called *1-cycle PDDL problem*, which corresponds to 1 iteration of the cyclic plan, with a standard temporal planner. The steady-state resulting in the minimal makespan plan is saved.
5. A plan to generate N instances of the product is generated by sequencing (a) a path from the initial state to the beginning of the first cycle (*setup phase*), (b) N unrolled iterations of the best cyclic plan, and (c) a path from the end of the last cycle to the final state where all products have exited (*cleanup phase*).

3.1 Difficulties in Identifying Steady States

A candidate steady-state (*SS*) for cyclic plans in the CELL-ASSEMBLY domain can be described in terms of the current

state of a set of (partially processed) bases e.g. "there are three bases at a table, painter and machine, and the second one has been painted." The corresponding SS, S_i, is a set of partially grounded state variables, e.g., {(at b_{i+2} table), (at b_{i+1} painter), (painted b_{i+1}), (at b_i machine)}. S_i corresponds to a 1-cycle PDDL problem $\Pi(S_i)$, where the initial state and the goal condition includes S_i and S_{i+1}, respectively.

Given such a representation, identifying a *good* SS is reduced to systematically enumerating and evaluating candidate states S, based on the quality (e.g. minimal makespan) of the plan of $\Pi(S)$. However, finding the best plan is not trivial because both too large and too small number of products in the system results in inefficiency. Finding the candidate SS's is also difficult. In principle, we could enumerate all states reachable from some initial state and test whether each such state is a feasible SS, but this is clearly impractical for any nontrivial problem instance.

One simple, possible approach is based on noting that in any feasible SS, there are 1 or more partially processed products (bases) placed somewhere in the assembly plant – in the example above, {(at b_{i+2} table), (at b_{i+1} painter), (at b_i machine)}. Based on the domain definition, we know all the possible "locations" e.g. table, painter, machine, and we know that we could place at most 1 base in each location. Thus, we could enumerate these 2^3 possibilities (whether a base is at each location or not). Also, we must enumerate the possibilities for other features of the steady state, such as the locations of the movable arm(s), as well as which bases, if any, are held by the arms, and the properties other than location such as "a base is painted or not".

There are two serious problems with this approach: First, it assumes that domain-specific semantics are hard-coded into the algorithm: e.g., all "locations" are labeled, arms "move", bases can be "painted", etc. This results in a fragile, domain-dependent system, and our main goal is domain-independence. Second, the number of candidate states enumerated is too large, especially considering that evaluation of each candidate SS may invoke a standard planner.

We now propose a fully automated, domain-independent approach for identifying candidate SS's in ACP. This approach does *not* assume, for example, that locations and movable objects are explicitly labeled in the domain model, nor does it require us to use domain-specific knowledge to constrain the search for good candidate SS's. Instead, our approach automatically extracts "places" and "movements" associated with partial products (bases), based on a structural analysis of the domain model and a template plan. Furthermore, these automatically extracted features can be used to significantly constrain the set of candidate steady-states, as shown in the later section.

Typed Predicates In describing our methods, we use the following notation of *Typed Predicates*. In PDDL domains with :typing requirements, a type can be assigned to each object. If no type is specified, it is defaulted to a predefined type `object`, which we abbreviate as $\star$ (don't-care). We denote *type* $(o) = \tau$ when o is of type τ, where o is either an object or a parameter (of predicates and actions). Types can have a hierarchy, such as "type A is a *subtype* of B", which we denote by $A \leq_\tau B$. In turn, B is a *supertype* of A. Also, we write $o_1 \leq_\tau o_2$, when *type* $(o_1) \leq_\tau$ *type* (o_2) for objects o_1 and o_2. A *typed predicate* is a predicate whose parameters are specialized to some type, including $\star$. We distinguish between two typed predicates which have the same name but are specialized to the different types. For two typed predicates p_1 and p_2 with the same name, we say p_1 *completely specializes* p_2 when:

$$k = 1 \text{ or } 2, \langle v_{ki} \rangle = \textit{params}\,(p_k)\,, \forall i;\ v_{1i} \leq_\tau v_{2i}$$

where *params* (p) is a parameters of a predicate p, and we denote this by $p_1 \leq_\tau p_2$. Note that we also use *params* (a) to suggest the parameters of an action a. Also, (pred a b) means an ungrounded typed predicate with parameters specialized to type a,b .

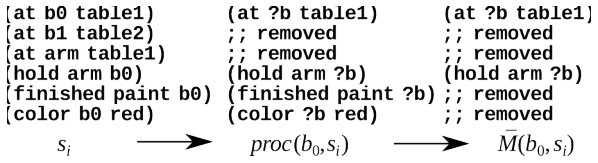

Figure 3: A state, its corresponding *process* and *movement*

3.2 Building and Analyzing a Plan Template

As stated in Sec. 3, ACP takes a PDDL domain, a type τ, a number N and a problem. The problem may contain $n \geq 1$ instances of a single product, but n should be small so that a good plan is obtained. We solve the problem with a standard planner and get a plan P, which we call as a "plan template". We arbitrarily choose one object b_0 of type τ in the problem. The first major step is here: we identify the "processing steps" of b_0. This is formally defined as follows:

Definition 1 (Process)**.** *proc* (b_0, s_i), the *process* for a product b_0 in i-th state s_i that appears during the execution of P is the subset of propositions $\{f \in s_i \mid b_0 \in \textit{params}\,(f)\}$ in s_i such that every occurrence of b_0 has been replaced with a variable b.

Definition 2 (Whole Processes)**.** The *Whole Processes*, *proc** (b_0), of a product b_0 in P is the sequence of *proc* (b_0, s_i) for all state s_i in P.

For example, the first step of Fig. 3 gives an example of computing *proc* (b_0, s_i) from some s_i. In effect, *proc* (b_0, s_i) removes all propositions from s_i that do not involve b_0. By applying this procedure to every step of the plan template, we extract *proc** (b_0), which captures the flow of a base as it progresses through the plan.

The next major step is to automatically extract things that correspond to "places" and "movements". Intuitively, a "place" is occupied by the product (base). This means that place-related predicates must involve the base, and *proc** (b_0) allows us to identify only those predicates involving a base. In Fig. 3, given *proc* (b_0, s_i) a human can infer that (at ?b table1) indicates that a base is located at "table1", which means that "table1" is a "place". On the other hand, in Fig. 3, although (color ?b red) has the same syntactic structure as (at ?b table1), i.e., (predicate ?b symbol), "red" is

not a location. ACP must correctly automatically infer that "table1" is a place, but "red" is not, but without access to human-level understanding of natural language and commonsense knowledge.

Owner/Lock Predicates The key difference between "place" and "non-place" is the implication of (or lack thereof) a particular kind of mutual exclusion relationship. Note that (at b_1 table1) is not just a statement about the location of b_1. It also implies something about the occupancy of table1, i.e., if b_1 is at table1, then it is occupied by b_1. In this case, it is a resource with capacity 1, and can be treated as a mutex resource. *If all "places" have unit capacity, any time a product moves into a "place", it must grab a lock on that place.* Otherwise, the model would allow multiple products to simultaneously occupy a single place.

Therefore, we can distinguish places from non-places by identifying this kind of mutex relation in the domain model and the template plan. In Fig 3, "color" and "finished" do not impose such mutex constraints because there is no limit on the number of objects that can be simultaneously assigned the color "red", or be "finished" with the "painting" step.

How are mutex constraints indicating a "place" represented in PDDL? Consider the following: *When base b_0 satisfies some condition, no other base can satisfy the same condition*. We can directly model these constraints in the expressive ADL subset of PDDL. Let the condition be (at b_0 table). If B is the set of all bases and the add effect of some action a includes (at b_0 table), a must have a precondition that implies the following constraint:

$$C_1 : \forall b \in B \setminus \{b_0\} ; \neg(\text{at } b \text{ table})$$

Since most current domain-independent planners support only a limited subset of PDDL (e.g., STRIPS+α) they do not directly support "forall", and constraints such as C_1 are usually implemented using a predicate which represents mutually exclusive use of "table", e.g., (table-occupied ?table). We call them mutual exclusion predicates or *lock predicates*. Also, if we view (table-occupied table) as a "lock", then we can interpret (at b_0 table) as an "ownership" predicate indicating that base b_0 now holds the lock.

The introduction of such predicates is very common in practice, and at least in STRIPS, this is by far the most common pattern in STRIPS for expressing C_1; If another pattern exists, it may still be possible to add a new lock/owner detection technique, as further discussed in Sec. 5.

It seems possible to mechanically identify lock/owner pairs by a *domain-independent* structural analysis of the action schema because all owner/lock pairs appear in the similar manner in all action definitions. For example, an action a that adds (at b_1 table) as an effect will have (not (occupied table)) as a precondition. Also, if base b_1 is currently at another location, e.g., (at b_1 X) is a precondition of a, then there will be a "lock" corresponding to X, e.g., (occupied X), which will be a precondition of a, and will also be in the delete effect of a.

These predicates are useful for 2 reasons: (1) they allow us to enumerate the sets of objects that represent "places" and therefore all possible assignments of products on such places; (2) assignments that violate mutex constraints can be eliminated from consideration. More generally, we will identify pairs of *lock* and *owner* predicates from the *proc** (b_0) of a template plan, and we can define a candidate steady-state as an assignment of 0/1 values to the owner predicates.

Detection Method We now describe how our owner/lock detection mechanism works with a simple example. Suppose we model a 2-D grid of cells that can be occupied by objects. We need to infer that (2d x b y) with types (2d coord base coord) and (occupied x y) with (occupied coord coord) is a lock/owner pair (coord means "coordinate".) Note that *the names of parameters are not considered at all* during the detection – only the types. We use names x, y, base only for the reader's convenience.

First, since the corresponding parameters x,y in 2d and occupied appear at the different positions in the parameters, a mapping between them, from the mutex lock μ to the owner o, must be identified. The mapping is denoted by $\pi = \{(j \to i) \ldots\}$ where each $(j \to i)$ maps μ_j to o_i (the j, i th parameter of μ, o.) occupied/2d has $\pi = \{(0 \to 0), (1 \to 2)\}$ i.e. $(0 \to 0)$ maps x in occupied to x in 2d. Since we do not consider the parameter name, $\{(0 \to 2), (1 \to 0)\}$ is also valid (it maps x to y and y to x.)

Next, we checks if each $(j \to i)$ satisfies a type relationship $o_i \leq_\tau \mu_j$ (or discard it otherwise.) If they have no inheritance (e.g. type orange for o_i and apple for μ_j) then no instance satisfies those types at once and it does not provide reasonable information. Being μ_j the supertype is mandatory (described next.) occupied/2d with π passes this check.

Third, we must ensure that (2d x b y) implies (occupied x y). It obviously requires $o_i \leq_\tau \mu_j$. Also, it requires the following conditions to hold in any actions:

1. When occupying a place, ensure the place is not in use, and acquire the lock.
2. When leaving the place, release the lock.

For example, we check for any action such that (1) if it adds (2d x b y), it has (not (occupied x y)) in the precondition and it adds (occupied x y) at the same time, and (2) if it deletes (2d x b y), it also deletes (occupied x y).

We enumerate all such possible pairs of typed predicates by trying all the possible parameter supertypes for two predicates. For example, the possible parameter supertypes of 2d may be (* * *), (coord * *), (* base coord) etc. Since the size of a PDDL domain is usually small, the time required for this exhaustive enumeration is inconsequential.

When there are two such valid owner/lock pairs, where two owners share the name of the predicate, as do the locks, and one completely specializes the other, then the more specific one is discarded. For example, given 2 pairs of owner/lock with types ((2d coord * coord) , (occupied coord coord)) and ((2d coord base coord) , (occupied coord coord)) , only the former is kept.

Finally, we note that an additional mechanism should be applied during the checks over the actions. In PDDL, (1) the predicates in the action definition are described with the parameters of the action and (2) the types of these action parameters are independent of the predicate definitions.

Thus, while we are testing if a pair satisfies the lock/owner relationships by scanning over the actions, we must check if (1') the parameters of the predicates in an action follow the mapping which the original owner/lock pair has and (2') the predicates in an action matches the types of owner/lock pair. Otherwise, the pair of predicates that appeared in an action is *not* an instance of the owner/lock pair we are checking. Consider the following example: here are two actions that *do not* match an owner $o =$ (2d coord base coord) and a lock $\mu =$ (occupied coord coord) with a mapping $\pi = \{(0 \to 0), (1 \to 2)\}$ because ex1 follows the different mapping $\{(0 \to 2), (1 \to 0)\}$ and ex2 does not match the type relationships. In both cases, the checks should not be applied to these instances, and they should not affect the validity of $\langle o, \mu, \pi \rangle$.

```
(:action ex1 :parameters (?x ?y - coord ?b - base)
 :precondition (not (occupied ?x ?y))
 :effect       (and (2d ?y ?b ?x) ... ))
(:action ex2 :parameters (?x - orange ?y ?b - apple)
 :precondition (not (occupied ?x ?y)) ...)
```

Now we formalize the above notion as follows. We assume :negative-preconditions, because it makes the definitions of "locks" and "owners" straightforward. For STRIPS, there is an analogous notion of a *releaser* which, instead of modeling the "lockedness" of a resource, directly models the converse – that a resource is "available", e.g., (free arm) instead of (unavailable arm). While this is implemented in ACP, we omit the explanation both for space and clarity.

First, let the domain $\mathcal{D} = \langle \mathcal{P}_\tau, \mathcal{A} \rangle$ where $\mathcal{P}_\tau$ is the set of typed predicates and $\mathcal{A}$ the operator set. Also, $o, \mu \in \mathcal{P}_\tau$, $\mathbf{o} = \mathit{params}(o) = \langle o_i \rangle$ and $\mathbf{m} = \mathit{params}(\mu) = \langle m_j \rangle$. Assume $|\mathbf{o}| \geq |\mathbf{m}|$. Also, let $p_{\mathbf{x}}$ mean an application of a predicate p to parameters $\mathbf{x}$.

Definition 3 (Mapping of Parameters)**.** $\langle o, \mu \rangle$ has a *mapping of parameters* π when it is a one-to-one projection $\pi : j \mapsto i$ s.t. $\forall m_j \in \mathbf{m}; i = \pi(j) \Rightarrow o_i \leq_\tau m_j$.

Definition 4 (Matching Criteria in Action Definition)**.** Assume $\langle o, \mu \rangle$ has a mapping π. Let a an action, where $\mathit{params}(a) \supseteq \mathbf{x} \supseteq \mathbf{y}$. Then $\langle o_{\mathbf{x}}, \mu_{\mathbf{y}} \rangle$ *match* $\langle o, \mu, \pi \rangle$ when:

$$\forall j; (y_j = x_{\pi(j)}) \wedge (y_j \leq_\tau \mu_j) \quad \text{and} \quad \forall i; x_i \leq_\tau o_i \ .$$

Definition 5 (Owner-Lock relationship)**.** We say o and μ are in a *Owner-Lock relationship* when $\langle o, \mu \rangle$ has a mapping π and, $\forall a \in \mathcal{A}$, when $\langle o_{\mathbf{x}}, \mu_{\mathbf{y}} \rangle$ match $\langle o, \mu, \pi \rangle$, both the followings hold:

$$o_{\mathbf{x}} \in e^+(a) \Rightarrow \mu_{\mathbf{y}} \in e^+(a) \wedge \tilde{\mu_{\mathbf{y}}} \in \mathit{precond}(a)$$
$$o_{\mathbf{x}} \in e^-(a) \Rightarrow \mu_{\mathbf{y}} \in e^-(a)$$

where $e^+(a)$, $e^-(a)$ and $\mathit{precond}(a)$ is the add effect, delete effect and precondition of a, and $\tilde{\mu}$ is a negative precondition (not μ).

Movement Returning to Fig. 3, we finally have a method to extract only the "change of the place" or *Movement* in $\mathit{proc}^*(b)$ by filtering the owners in each $\mathit{proc}(b, s_i)$ and remove the unchanged (set-equal) part. The figure shows that our method correctly filters "non-places" out, such as finish and color. The formal definition follows:

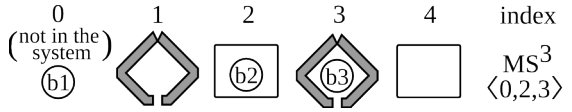

Figure 4: Example "Movements-Simplified Steady States"(MS^3)

Definition 6 (Movement)**.** Let O be the set of owner predicates. For a product b in a template plan, for i-th state s_i that appears during the execution of P, Movement $\bar{M}(b, s_i)$ is:

$$\bar{M}(b, s_i) = \{f \in \mathit{proc}(b, s_i) \mid \exists o \in O; f \leq_\tau o\}$$

Definition 7 (Whole Movements)**.** We form *Whole Movements* $\bar{M}^*(b)$ by enumerating all $\bar{M}(b, s_i)$ in a template plan P and iteratively removing one of each pair of movements that are adjacent and set-equal to each other.

The fact that (hold arm ?b) in Fig. 3 is a "location" may be confusing: what happens if the arm moves? Would this not imply that the position of ?b is also changed? The answer is no – what matters is the *resource usage* in the system, and *not* where the spatial location of the resource, e.g., changing the arm position does not affect whether the gripper on the arm is occupied by a base or not.

3.3 Enumerating and Filtering the Steady States

Based on a sequence of Movements $\bar{M}^*$, we can represent a candidate SS as a set of indices $\{i_0, i_1, \ldots i_{k-1}\}$ (See Fig. 4) where k is a number of partial products in a cycle. Each number represents which set of owners in $\bar{M}^*$ a partial product has at the beginning of a cycle. Note that the indices $i_0 = 0$ and $i_{k-1} = \left|\bar{M}^*\right|$ represent the states "not yet in the system" and "already out of the system", respectively, and the corresponding partial products occupy no locks. We call this representation an MS^3, which stands for "Movements-Simplified Steady States". We enumerate all feasible MS^3 which satisfy the mutex constraints by adding a new number to the already checked MS^3, initially $\{0\}$. There are potentially $2^{|\bar{M}^*|-1}$ candidate SS's (the first element $i_0 = 0$ is fixed).

In theory, we could enumerate each of these candidates and identify the best SS, but brute-force evaluation of all $2^{|\bar{M}^*|-1}$ candidates (with a standard planner) is impractical, both because the difficulty of each 1-cycle problem increases with k and because large $2^{|\bar{M}^*|-1}$ can be intractable. Therefore we applied some filtering methods on these candidates.

Mutex Focused Planning Our main filtering method is called *Mutex Focused Planning* which removes all candidate steady-states from which there is no path (plan) to the beginning of the next cycle, due to unsatisfiable mutex constraints (e.g., deadlocks, resource starvation). For example, in the CELL-ASSEMBLY domain, consider a candidate MS^3 which puts products on all possible locations. This results in deadlock, because all arms, tables and machines are occupied and no base can move.

There might be various methods to detect such deadlocks but we chose a simple Dijkstra search to reduce

the implementation effort. In this search, each state is a MS^3, e.g., $\{0, 2, 5\}$. Its successor states can be obtained by incrementing one of the elements of the MS^3, i.e., $\{1, 2, 5\}, \{0, 3, 5\}, \{0, 2, 6\}$. However, some of these successor states may violate the mutex constraints and they should be discarded.

This can be interpreted as an abstraction of the original problem represented by the steady-state, which only considers "which product moves in what order" and abstract away the details of each move. Each transition represents a movement of one product from a place to the next place (as determined by the template plan). Also, the mutex constraints prevent each product from moving into a place occupied by another product.

When a SS is represented by MS^3 $\{0, i_1, \ldots, i_{k-1}\}$ and it has a path to $\{i_1, \ldots i_{k-1}, |\bar{M}^*|\}$ in the search space described above, then we say that it has a *mutex-feasible path* (MFP). We remove all SS's with no MFP. There may be some SS's whose corresponding 1-cycle problems have no solutions because of the factors other than mutex constraints, but we assume they are detected in the subsequent call to the standard planner (see Sec. 3.4).

Filtering Heuristics Even after eliminating all candidate SS's without MFP, we further reduce the set of candidates with filtering heuristics. We identify groups of "similar" SS's which are likely to have the same MFP. We instantiate and fully evaluate only the first instance of such a group, discarding the rest of the members. Based on the fact that any point on a cyclic path can be a start of the path, we have:

Theorem 1. When a MS^3 has a MFP $S_{I_1} = \{0, i_1, \ldots, i_{k-1}\} \rightarrow S_{I_2} = \{0, j_1, \ldots j_{k-1}\} \rightarrow S_{G_1} = \{i_1, \ldots i_{k-1}, |\bar{M}^*|\}$, then S_{I_2} also has a path $S_{I_2} \rightarrow S_{G_2} = \{j_1, \ldots j_{k-1}, |\bar{M}^*|\}$.

Proof. For each point S of a MFP $S_{I_1} \rightarrow S_{I_2}$, if we remove 0 from S and add $|\bar{M}^*|$ to S, then we get a path $S_{G_1} \rightarrow S_{G_2}$. Stringing $S_{I_2} \rightarrow S_{G_1}$ and $S_{G_1} \rightarrow S_{G_2}$ yields a path $S_{I_2} \rightarrow S_{G_2}$. This manipulation always yields a MFP because $|\bar{M}^*|$'th and 0'th movement has no lock (they represent the states of "out of the system".) □

Similarly, there is also a path $S_{I_2} \rightarrow S_{G_2}$ for all S_{G_2} in $S_{I_1} \rightarrow S_{G_2} \rightarrow S_{G_1}$ (details omitted due to space).

3.4 Planning the Cycles Based on Steady States

After reducing the number of SS's, we build and solve a corresponding *1-cycle PDDL problem* $\Pi(S)$ for each SS S. The initial state of $\Pi(S)$ consists of predicates that either (1) describe the initial state of each partial product in SS, or (2) describe the global initial state. To construct (1), we find corresponding *proc* (b, s_i) for each j in MS^3 $\{\ldots j \ldots\}$ and ground it with a product of an arbitrary name, e.g. in Fig. 3, we substitute ?b in *proc* (b_0, s_i) with new-b_j. Note that i and j may differ because identical adjacent elements are removed in $\bar{M}^*$. (2) is excerpted from the initial state of the template problem – only those predicates that do not have a product in its arguments such as (at arm table1) are chosen. Additionally, (3) appropriate lock predicates are added, such as (occupied table1),(holding arm). The goal state construction is similar and straightforward. We solve each $\Pi(S)$ with a standard domain-independent planner. We choose the minimal makespan plan, store the corresponding SS and unroll the plan for an arbitrary number of products N.

In addition to the 1-cycle problem, we also need to plan the setup that takes us from the initial state to the beginning of the cycle, and a cleanup that takes us from the end of the last cycle to the final state. These are straightforward and not described here due to space. Finally, we concatenate them and get the whole solution of N products.

ACP calls a standard domain-independent planner in order to solve these problems (the same planner is used to generate the template plan, and to solve the setup/cleanup problems). Ideally, a temporal planner should be used when the objective is to minimize makespan. However, due to difficulties finding a robust temporal planner that could reliably handle all of the subproblems generated by ACP, we currently use Fast Downward [Helmert, 2006] with the LAMA2011 emulation configuration to generate a sequential plan, which is then parallelized by applying a simple scheduler using the minimum-slack algorithm of [Smith and Cheng, 1993]. Actions in the input model are treated as durative actions with the commonly used "over-all" semantics, where the preconditions of an action are interpreted to be true at the beginning of an action as well as throughout the action [Cushing et al., 2007].

4 Experimental Evaluation

In this section, we evaluate and compare ACP with (1) direct application of standard domain-independent planners (2) a simple cyclic planning method which concatenates plans generated with domain-independent planners as well as (3) lower bounds computed by several methods. In all experiments below, ACP is executed on an Intel Xeon E5410@2.33GHz with a total time limit of 60 minutes. Each evaluation of a candidate SS (solving the steady-state planning problem instance) is limited to 240[s]. In order to compare ACP with the other planners, we limited the number of SS's under 50. This eventually resulted in shorter total computation time than 60 minutes in all benchmark problems, including setup/cleanup planning.

Our evaluation of ACP is primarily based on instances of the CELL-ASSEMBLY domain. We used 5 instances of the CELL-ASSEMBLY: Instance 2a (shown in Fig. 2), 2b, 3a, 3b, 3c (most difficult). For each problem above, manufacturing orders for $4, 16, \ldots$ **1024** product instances were generated and then solved by ACP. All benchmark domains are available at http://guicho271828.github.io/publications/ .

4.1 Comparison with the Direct Application of Standard Planners

We first investigate the performance of standard domain-independent planners on our benchmark. We evaluated: (1) Fast Downward (FD) using the LAMA2011 configuration, using our postprocessing scheduler to parallelize the plan, (2) Fast Downward with the Landmark Cut heuristic + post-

processing scheduler, (3) CPT4 [Vidal, 2011a], an admissible temporal planner (using the h^2 heuristic), and two non-admissible temporal planners (4) yahsp [Vidal, 2011b] and (5) $\text{DAE}_{\text{yahsp}}$ [Dréo et al., 2011]. All planners were executed with a 6 hour time and 4GB memory per instance, for each problem size. For the temporal planners, all domains were converted to temporal domains with :durative-actions.

For each problem, Table 1 shows the largest $N \in \{4, 16, 64, 256, \mathbf{1024}\}$ such that a manufacturing plan could be generated by each of the above planners. For example, on model2a, $\text{DAE}_{\text{yahsp}}$ was able to solve the problem $N = 4$ with the least makespan of the 5 configurations, and none of them were able to solve problems with the number of products $N \geq 16$. This shows that current domain independent planners (both temporal planners such as CPT/yashp/DAE, as well as sequential planners such as FD, with postprocessing) can only solve small-scale manufacturing problems to assemble relatively few instances of the desired product.

4.2 Comparison with "Simple Cyclic Planning"

A simple method for generating cyclic plans to manufacture N instances of a product is: Use a domain independent planner to generate a parallel, temporal plan for K instances, where $K < N$, and then repeat the K-instance SCP-template plan $\lfloor N/K \rfloor$ times, followed by a plan to generate the rest of $N - (K \times \lfloor N/K \rfloor)$ "remainder products". In order to be able to concatenate these K-instance SCP-template plans, the goal state specify that in addition to assembling K products, the state of the plant must return to the same state as in the initial state (e.g., in the CELL-ASSEMBLY domain, the robot arms must be in the same position as in the initial state).

We compare the above *Simple Cyclic Planning* (SCP) algorithm with ACP. For each problem, SCP-template plans are generated for $1 \leq K \leq 9$ products. For each K, SCP was given a time limit of $t = 1333$[s] and 2[GB] memory, for a total of 12000[s] per problem.[1] For each problem, we choose the best K-instance SCP-template plan which minimizes $(\text{K-product makespan})/K$. This was repeated for each of the 5 planners/configurations described above. Due to space constraints, we cannot show all results. Instead, for each problem, we identified the solver with minimum makespan per product. Thus, makespan$/K$, K and the corresponding solvers are shown in Table 1. *For each problem, the best SCP result (out of 5 planner configurations), each given 12000 seconds, is shown.* In comparison, the ACP results are the result of a single run given 12000 seconds.

For small orders ($N = 4$ orders), the ACP makespan is comparable to the SCP makespan, but for large orders ($N \geq 64$), ACP clearly outperforms SCP. The relative inefficiency of ACP in the small orders can be ascribed to the setup/cleanup phases. In small orders, the impact of the cyclic efficiency can be sometimes canceled by the inefficiency in the setups. So far, we have made no effort to optimize the setup/cleanup phases, and this remains an avenue for future work. However, for large N (which is the motivation for investigating cyclic planning in the first place), setup/cleanup costs are amortized, and the cost per product of ACP is significantly better than that of SCP.

[1] We ignore the planning time for the "remainder products" because we want to focus on the cyclic nature of the problem.

Problem	$M = \lvert\bar{M}^*\rvert$ (♯ Movements)	(a) 2^{M-1}	(b)	(c)	(d)	(e)
CELL-ASSEMBLY 2a	20	$\approx 5.3 \times 10^6$	104471	6538	412602	677
CELL-ASSEMBLY 2b	10	512	143	85	266	18
CELL-ASSEMBLY 3c	40	$\approx 5 \times 10^{11}$	N/A	N/A	N/A	N/A
Woodworking	6	32	1	21	0	10
Barman	9	256	240	13	0	3

Table 2: All SS's (a) obtained via $\bar{M}^*$, (b) filtered by the start state feasibility, (c) pruned by the filtering heuristics, (d) filtered by mutex focused planning, (e) remained after (b,c,d). Here, $e = a - (b + c + d)$ holds. For problem 3a, 3b and 3c, we failed to compute (e) even with the aid of filtering heuristics, because the number of total SS's is too large. (This does not matter during the search because each SS is incrementally instantiated as needed.)

4.3 Comparison with Lower Bounds

To assess how close the solutions found by ACP are to an optimal cyclic plan, we manually computed lower bounds (shown in Table 1) for the makespan of 1 cycle for each problem instance, based on straightforward analysis of the bottlenecks in each problem. We also show bounds computed using CPT4, but these were mostly less accurate than our manually computed lower bounds. Comparing the highest (better) bounds of the two to the makespan found by ACP, the gap is between a factor of 1-4 on all instances except for Barman (gap of 7x).

4.4 Evaluation of the Pruning Methods

Table 2 shows the number of candidate steady-states that remains after each step in Sec. 3.3 is applied. This shows that for large problems (problems with large M), our pruning/filtering methods are quite effective in reducing the number of candidates to a manageable number.

4.5 Domain Independence of ACP

To demonstrate the domain-independence of ACP, we conducted experiments based on two "manufacturing" domains in IPC benchmarks.

Woodworking is based on the temporal domain of IPC'11, and the task is to cut and process wooden parts from large boards. Since ACP currently uses a non-temporal planner (Sec. 3.4), durative actions d were split into 2 actions, d-start and d-end, with additional predicate which ensures that d-end is always applied after d-start. In addition, while the original Woodworking limits the number of pieces that can be cut from a single piece of wood, we eliminate this constraint so that an arbitrary number of pieces can be cut (in effect, we assume that fresh boards are provided whenever the current board is exhausted). $\tau =$ part in this domain.

In the Barman domain, a 2-armed robot uses a shaker and a shot-glass to mix cocktails. The domain was used without any modification from the IPC benchmark domain. However, while IPC *instances* of this domain require the robot to

		ACP			SCP (best of 5 solvers)			Lower Bounds				Standard Planner				
Problem	♯ of products	run-time	makespan	makespan (per product)	makespan (per product)	♯ of products	solver	manual bound	manual bound	CPT(h2)	gap (ACP / bound)	FD/LM$_{\text{cut}}$ + scheduler	FD/LAMA + scheduler	yahsp	DAE	CPT4
	N	[sec]	c_{ACP}	c_{ACP}/N	c_{SCP}/K	K		l_m/N	l_m	l_{CPT}	$c_{\text{ACP}}/\max.l_x$					
CELL-ASSEMBLY	4	1048	**331**	82.8	83	2	FD/LM$_{\text{cut}}$	39	156	*176.3*	1.9	fail	892	807	*774*	fail
2a	16	1049	**1255**	78.4	↑	↑	+ scheduler	↑	*624*	460	2.0	fail	fail	fail	fail	fail
(2 arms, 5 jobs)	64	1049	**4951**	77.4	↑	↑	↑	↑	*2496*	1624	2.0	fail	fail	fail	fail	fail
(τ = base)	256	1049	**19735**	77.1	↑	↑	↑	↑	*9984*	fail	2.0	fail	fail	fail	fail	fail
	1024	1050	**78871**	**77.0**	↑	↑	↑	↑	*39936*	fail	2.0	fail	fail	fail	fail	fail
CELL-ASSEMBLY	4	34	**246**	61.5	62.3	3	FD/LM$_{\text{cut}}$	42	168	*181.12*	1.4	*249*	256	607	332	fail
	16	33	**978**	61.1	↑	↑	+ scheduler	↑	*672*	593	1.5	fail	fail	fail	fail	fail
2b (1a, 5j)	64	33	**3906**	61.0	↑	↑	↑	↑	*2688*	2241	1.5	fail	fail	fail	fail	fail
	256	34	**15618**	61.0	↑	↑	↑	↑	*10752*	fail	1.5	fail	fail	fail	fail	fail
	1024	35	**62466**	**61.0**	↑	↑	↑	↑	*43008*	fail	1.5	fail	fail	fail	fail	fail
CELL-ASSEMBLY	4	1893	**660**	165	171	1	FD/LAMA	44	176	*237*	2.8	fail	*1080*	fail	fail	fail
	16	1953	**2352**	147	↑	↑	+ scheduler	↑	*704*	345	3.3	fail	fail	fail	fail	fail
3a (3a, 10j)	64	1961	**9120**	142.5	↑	↑	↑	↑	*2816*	1257	3.2	fail	fail	fail	fail	fail
	256	1746	**36192**	141.4	↑	↑	↑	↑	*11264*	fail	3.2	fail	fail	fail	fail	fail
	1024	1973	**144480**	**141.1**	↑	↑	↑	↑	*45056*	fail	3.2	fail	fail	fail	fail	fail
CELL-ASSEMBLY	4	1163	**318**	79.5	81.3	3	FD/LM$_{\text{cut}}$	28	112	*191*	1.7	fail	*540*	715	fail	fail
	16	1162	**1074**	67.1	↑	↑	+ scheduler	↑	*448*	240	2.4	fail	fail	fail	fail	fail
3b (4a, 8j)	64	1163	**4098**	64.0	↑	↑	↑	↑	*1792*	897	2.3	fail	fail	fail	fail	fail
	256	1164	**16194**	63.3	↑	↑	↑	↑	*7168*	fail	2.3	fail	fail	fail	fail	fail
	1024	1165	**64578**	**63.1**	↑	↑	↑	↑	*28672*	fail	2.3	fail	fail	fail	fail	fail
CELL-ASSEMBLY	4	1968	**804**	201	203	1	FD/LM$_{\text{cut}}$	43	172	*335*	2.4	fail	*947*	fail	fail	fail
	16	1856	**2508**	156.8	↑	↑	+ scheduler	↑	*688*	532	3.6	fail	fail	fail	fail	fail
3c (5a, 11j)	64	1847	**9324**	145.7	↑	↑	↑	↑	*2752*	2068	3.4	fail	fail	fail	fail	fail
	256	1890	**36588**	142.9	↑	↑	↑	↑	*11008*	fail	3.3	fail	fail	fail	fail	fail
	1024	1894	**145644**	**142.2**	↑	↑	↑	↑	*44032*	fail	3.3	fail	fail	fail	fail	fail
Woodworking	4	11	**80**	20	17.2	9	FD/LAMA	15	60	*80*	1	**80**	**80**	150	**80**	**80**
(plane,grind,	16	11	**260**	16.3	↑	↑	+ scheduler	↑	*240*	185	1.1	**260**	330	590	270	fail
vanish)	64	12	**980**	15.3	↑	↑	↑	↑	*960*	665	1.0	**980**	1290	2170	12840	fail
(τ = part)	256	12	**3860**	15.1	↑	↑	↑	↑	*3840*	2585	1.0	fail	fail	fail	fail	fail
	1024	15	**15380**	**15.0**	↑	↑	↑	↑	*15360*	fail	1.0	fail	fail	fail	fail	fail
Barman	4	331	35	8.8	**6.3**	4	FD/LM$_{\text{cut}}$	1	4	*21*	1.7	fail	**23**	81	31	fail
	16	332	179	11.2	↑	↑	+ scheduler	↑	16	*26*	6.9	fail	fail	fail	fail	fail
(τ = shot)	64	332	755	11.8	↑	↑	↑	↑	*64*	fail	11.8	fail	fail	fail	fail	fail
	256	332	3059	11.9	↑	↑	↑	↑	*256*	fail	11.9	fail	fail	fail	fail	fail
	1024	332	12275	12.0	↑	↑	↑	↑	*1024*	fail	12.0	fail	fail	fail	fail	fail

Table 1: Comparison of ACP, SCP, lower bounds (manually computed bound and CPT h_2 bound), and direct application of standard planners for assembling 4, 16, 64, 256, 1024 products. SCP always returns the same cyclic schedule regardless of N (" ↑ " means "same as above".) CPT4 failed to compute the heuristic value in almost all large problems.

fulfill mixed orders that involve several different cocktails, our test instances require many instances of a single cocktail to be mixed. τ = shot in this domain.

As shown in Table 1, ACP can solve both of these domains. For Woodworking, when N is large, ACP generates significantly better results than SCP and direct application of standard planners. On the Barman domain, the cyclic plan generated by ACP is less efficient than the SCP cyclic plan – this is due to a peculiarity of the Barman domain which makes it ideally suited for SCP (described in the next section).

5 Discussion

ACP is "sound but incomplete". As already mentioned in Sec. 3.2, we can add *other* owner/lock detection methods to ACP, and some unforeseen models can cause potential lock/owners to be missed, so ACP is "incomplete" in this sense. Nevertheless, ACP is sound, in the sense that any plan generated by ACP are valid because ALL elements missed (due to the aforementioned incompleteness) in the template plan are recovered in the 1-cycle problem sent to the underlying planner (Fast Downward).

Suppose the capacity of a certain place is ≥ 2, e.g. a table has 2 slots slot1 and slot2. A single-product plan in such a domain contains a state like (at table1 base slot1), which ACP correctly detects as an owner. Since ACP only considers owners which appeared in a single product plan, slot2 is not utilized in MFP (reachability analysis). However even in such cases, any 1-cycle problem sent to the underlying planner contains the information about slot2 in the "global initial state" described in Sec. 3.4, and the planner takes it into consideration and return a valid plan. The only drawback is the potential parallelism (=plan quality) lost by ignoring the second slot. Barman is one such domain because a shaker can carry beverages twice an amount of a shot, and in fact, ACP

performed worse than SCP on Barman.

More generally, ACP (1) provides fast reachability analysis and (2) can generate efficient plans by exploiting parallel actions. Failing to identify owner/locks only reduces the number of products processed in parallel. Indeed, in a domain with no owner/lock, ACP returns a cyclic plan with no parallelism.

6 Related Work

Cyclic scheduling for robot move sequencing in robotic cell manufacturing systems, has been studied extensively in the OR literature [Dawande et al., 2005], and the general problem of cyclic scheduling has been considered in the AI literature as well [Draper et al., 1999]. Previous work focuses on algorithms for generating effective cyclic schedules for specific domains, and addresses the problem: Given the stages in a robotic assembly system, compute an efficient schedule. In contrast, our work addresses the problem: Given some PDDL model for assembling an instance of "something" (without any domain-dependent annotations), first identify the "stages" that could be used in a cyclic plan for mass manufacturing of the product, and then use these "stages" to generate a cyclic plan. Thus, comparison with domain-specific techniques is an area for future work.

ACP is somewhat related to macro abstraction systems such as Macro-FF [Botea et al., 2005], which automatically identifies reusable plan fragments. Macro systems strive to provide a very general abstraction mechanism, but are typically limited to relatively short macros (e.g., 2-step macros in Macro-FF). ACP, on the other hand, focuses on identifying lengthy "macros" (10-30 steps) to maximize parallelism on a limited class of domains.

The domain analysis in ACP is related in spirit to systems such as DISCOPLAN [Gerevini and Schubert, 1998] and TIM [Fox and Long, 1998]. While the current implementation of ACP requires type information to be provided for the domain-independent extraction of steady-states, much of the type information could be inferred automatically using a system such as TIM.

7 Conclusions

We described ACP, a domain-independent system for generating cyclic plans for "manufacturing" domains where the requirement is to generate many instances (up to thousands of units) of a single product. Generating more than a handful of instances of a product is beyond the capability of standard domain-independent planners. ACP overcomes this limitation with a novel, static domain analysis system which performs fully automatic generation of a cyclic plan for assembling many instances of a single product. We showed that ACP can effectively solve CELL-ASSEMBLY problems, and showed that it can be applied to other domains such as Woodworking and Barman domains.

While motivated by a factory cell-assembly application, ACP is domain-independent. Based on static analysis of the input PDDL model and a plan template for assembling 1 instance of a product (generated using a standard domain-independent planner), ACP automatically extracts all of the required structure. Other than the product's name in the template problem, no labels/annotation to the PDDL model are required, and no assumptions are made about the names of the types or other objects. ACP automatically infers how a (partially processed) product progresses through the system, and how viable candidates for the start/end states for cyclic planning can be generated.

Currently, there are significant constraints on the kinds of cyclic plans that can be generated by ACP. ACP assumes that all instances of the product progressing through the manufacturing plant will be processed in the exact same way, i.e., in the cyclic plan, each product instance is processed in the exact same order and manner. In addition, ACP currently does not allow mixed orders, e.g., "assemble N_1 instances of product P_1 and N_2 instances of product P_2". Relaxing these restrictions could enable more efficient usage of available resources and also make ACP applicable to a broader class of applications, as well as allow further exploitation of parallel actions.

References

Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2005. Macro-ff: Improving ai planning with automatically learned macro-operators. *J. Artif. Intell. Res.(JAIR)* 24:581–621.

Cushing, W.; Kambhampati, S.; Talamadupula, K.; Weld, D.; and Mausam. 2007. Evaluating temporal planning domains. In *ICAPS*.

Dawande, M.; Geismar, H. N.; Sethi, S. P.; and Sriskandarajah, C. 2005. Sequencing and scheduling in robotic cells: Recent developments. *Journal of Scheduling* 8(5):387–426.

Draper, D.; Jonsson, A.; Clements, D.; and Joslin, D. 1999. Cyclic scheduling. In *Proc. IJCAI*.

Dréo, J.; Savéant, P.; Schoenauer, M.; and Vidal, V. 2011. Divide-and-evolve: the marriage of descartes and darwin. *Proceedings of the 7th international planning competition (IPC). Freiburg, Germany*.

Fox, M., and Long, D. 1998. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*.

Gerevini, A., and Schubert, L. K. 1998. Inferring state constraints for domain-independent planning. In *AAAI*, 905–912.

Helmert, M. 2006. The fast downward planning system. *J. Artif. Intell. Res.(JAIR)* 26:191–246.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL-the planning domain definition language. *The AIPS-98 Planning Competition Comitee*.

Ochi, K.; Fukunaga, A.; Kondo, C.; Maeda, M.; Hasegawa, F.; and Kawano, Y. 2013. A steady-state model for automated sequence generation in a robotic assembly system. *SPARK 2013*.

Smith, S. F., and Cheng, C.-C. 1993. Slack-based heuristics for constraint satisfaction scheduling. In *AAAI*, 139–144.

Vidal, V. 2011a. CPT4: An optimal temporal planner lost in a planning competition without optimal temporal track. *Proceedings of the 7th international planning competition (IPC). Freiburg, Germany* 25–28.

Vidal, V. 2011b. YAHSP2: Keep it simple, stupid. *Proceedings of the 7th international planning competition (IPC). Freiburg, Germany* 83–90.