4C1-3

Fully Automated Cyclic Planning for Large-Scale Manufacturing Domains

Masataro Asai Alex Fukunaga

Department of General Systems Studies
Graduate School of Arts and Sciences
The University of Tokyo

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.

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 13]. 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 99], the start and end states of the planning instance correspond to a "step forward" in an assembly line, 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. This paper[*1] proposes ACP (Automated Cyclic Planner), which fully automates the cyclic scheduling process for "mass manufacturing", e.g., cell assembly.

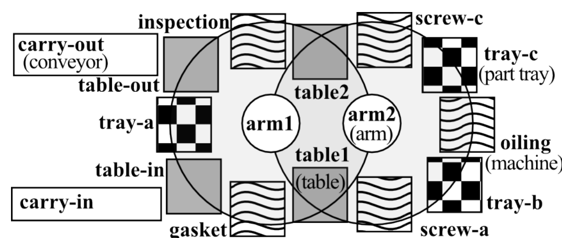

図 1: Example CELL-ASSEMBLY instance: model2a.

2. CELL-ASSEMBLY Domain

CELL-ASSEMBLY is a PDDL domain with STRIPS-style actions, negative-preconditions, action costs, and a hierarchical type structure.

In the CELL-ASSEMBLY domain, the task is to complete many products on an assembly line with robot arms. There are a number of assembly tables and machines that perform specific jobs such as painting a product or tightening a screw. In each assembly table, various kinds of `parts` are attached to a `base`, the core component of the product. For example, a problem requires two kinds of parts, `part-a,part-b`, to be attached to each one of `base0,base1`. The final products look like `base0/part-a/part-b` and `base1/part-a/part-b` .

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 13].

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 pre-

Contact: guicho2.71828@gmail.com

*1 Note that this is an extended abstract of an international conference paper [Asai 14]. Since much of the details are omitted due to space, please refer to the original paper for more information. The camera-ready copy is going to be public at `http://guicho271828.github.io/publications`.

conditions 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. It does not handle the mixed orders in which multiple types of products are assembled simultaneously.
- *Uniform Manufacturing Process:* To fulfill the order, all instances of product must be assembled in the exact same order using the exact same machines.
- *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.

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.

3.2 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 `*` (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 `*`. 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:

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

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 .

3.3 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 $\{f \in s_i \mid b_0 \in params(f)\}$ 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. 2 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,

```
(at b0 table1)             (at ?b table1)             (at ?b table1)
(at b1 table2)             ;; removed                 ;; removed
(at arm table1)            ;; removed                 ;; removed
(hold arm b0)              (hold arm ?b)              (hold arm ?b)
(finished paint b0)        (finished paint ?b)        ;; removed
(color b0 red)             (color ?b red)             ;; removed
```

s_i ⟶ $proc(b_0, s_i)$ ⟶ $\bar{M}(b_0, s_i)$

図 2: A state, its corresponding *process* and *movement*

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". In Fig. 2, 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.

3.3.1 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 occupy a single place.

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".) Though much of the detail is omitted here, essentially we checks if (2d x b y) implies (occupied x y). 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 formalize the above notion as follows. First, we assume :negative-preconditions, because it makes the definitions of "locks" and "owners" straightforward. 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, $o, \mu \in \mathcal{P}_\tau$, $\mathbf{o} = params(o) = \langle o_i \rangle$ and $\mathbf{m} = params(\mu) = \langle m_j \rangle$. 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 $params(a) \supseteq \mathbf{x} \supseteq \mathbf{y}$. 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:

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

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

3.3.2 Movement

Returning to Fig. 2, we finally have a method to extract only the "change of the place" or *Movement* in $proc^*(b)$ by filtering the owners in each $proc(b, s_i)$ 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:

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:

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

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. 2 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.

3.4 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. 3) 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).

Brute-force evaluation of all candidates to identify the best SS 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.

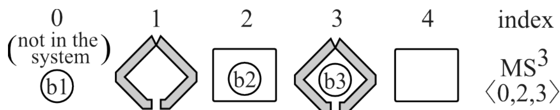

図 3: Example "Movements-Simplified Steady States"(MS^3)

3.4.1 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.

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.

3.4.2 Filtering Heuristics

Even after eliminating all candidate SS's without MFP, we further reduce the set of candidates, based on the fact that any point on a cyclic path can be a start of the path. We instantiate and fully evaluate only the first instance of such a group, discarding the rest of the members.

3.5 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. (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, currently Fast Downward [Helmert 06] with the LAMA2011 emulation configuration. 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.

4. 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.

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.

参考文献

[Asai 14] Asai, M. and Fukunaga, A.: Fully Automated Cyclic Planning for Large-Scale Manufacturing Domains, in *International Conference on Automated Planning and Scheduling* (2014)

[Draper 99] Draper, D., Jonsson, A., Clements, D., and Joslin, D.: Cyclic Scheduling, in *Proc. IJCAI* (1999)

[Helmert 06] Helmert, M.: The Fast Downward Planning System., *J. Artif. Intell. Res.(JAIR)*, Vol. 26, pp. 191–246 (2006)

[Ochi 13] Ochi, K., Fukunaga, A., Kondo, C., Maeda, M., Hasegawa, F., and Kawano, Y.: A Steady-State Model for Automated Sequence Generation in a Robotic Assembly System, *SPARK 2013* (2013)