Evaluating PDDL for programming production cells: a case study

Christoph Mayr-Dorn, Alexander Egyed
Johannes Kepler University
Linz, Austria
firstname.lastname@jku.at

Mario Winterer, Christian Salomon
Software Competence Center Hagenberg GmbH
Hagenberg, Austria
firstname.lastname@scch.at

Harald Fürschuß
ENGEL Austria GmbH
Schwertberg, Austria
harald.fuerschuss@engel.at

ABSTRACT

A unique selling point for cyber-physical production system manufacturers becomes the easy with which machines and cells can be adapted to new products and production processes. Adaptations, however, are often done by domain experts without in-depth programming know-how. We investigate in this paper, the implications of using a planning-based approach for using a domain expert's knowledge to control the sequences of a robot and injection molding machine (IMM). We find that current engineering support is insufficient to address testing, understanding, and change impact assessment concerns during the evolution of a PDDL/HDDL domain specification.

KEYWORDS

Robot programming, end-user programming, manufacturing automation, planning, symbolic AI, PDDL, HDDL

1 INTRODUCTION

A key enabler to achieving production of lot-size one at the cost of mass production are flexible production systems. Such flexibility implies that a manufacturer of a machine or production cell cannot foresee all future use cases. Subsequently, a key competitive advantage for a machine manufacturer becomes the easy with which machines and cells can be adapted to new products and production processes. This reprogramming of machines is often done by domain experts that are typically non-programmers who have product and production know-how but are not well versed in programming.

The problem for non-programmers becomes how to program a machine or production cell while covering a sufficient set of edge cases and situations to avoid frequent standstill. This quickly becomes infeasible as beyond some trivial use cases the resulting programs become unmaintainable: they are hard to understand, hard to reuse, and hence hard to evolve for non-programmers.

Domain experts, however, have detailed know-how of the production process, the stages the product needs to go through from raw parts and material to the intermediate and final states, i.e., the product's production life-cycle. To this end, they are knowledgeable about the preconditions of each of these steps and what the effects of the various stages are.

Thus the question emerges whether this know-how can be used as the basis for defining production cell behavior. We limit non-programmers to define (or reuse) preconditions and effects of actions, goals, and constraints, and have a planner determine what each machine should be doing at what point in time. In this paper, we investigate based on a case study whether PDDL (the The Planning Domain Definition Language) [11] and its extension for hierarchical planning problems HDDL [9] in combination with contemporary planners are a suitable approach for this.

Along these lines, we aim to answer two research questions (RQs): Specifically we want to obtain first insights into the practicality of applying such symbolic AI approaches for obtaining control commands in an injection molding cell, hence to what extend is PDDL/HDDL suitable finding optimal robot and machine sequences (RQ1). Additionally we ask to what extent adaptations are needed to extend the definitions of a basic injection molding cell for more advanced scenarios: hence to learn how complex adaptations of PDDL/HDDL specifications would become and thus what practical implications for end user programming this brings (RQ2).

We find, while PDDL/HDDL could be the basis for planning, they are currently impractical due to the following reasons. First, there is little support to obtain cyclic plans for producing more than just a few product instances. Second, plans are strictly sequential, thus when any, even minor, expected deviation occurs, replanning has to occur. And third, no solver we investigated is able to provide time-optimized plans beyond trivial problem instances within an acceptable amount of time.

The remainder of this paper is structured as follows: Section 2 introduces the investigated modeling languages and describes a typical scenario production cell. In Section 3 we detail how to encode the actions and constraints in our case study and some modeling alternatives before we evaluate in Section 4 the runtime behavior for solving various production scenarios. We then lay out in Section 5 the benefits and drawbacks as well as practical implications of using PDDL or HDDL. We discuss related work in Section 6 before concluding the paper with an outlook on future work in Section 7.

2 MOTIVATION AND BACKGROUND

2.1 Industry Context

ENGEL is a world leader in manufacturing injection molding machines (IMM) ENGEL also offers several industrial robots which are usually delivered together with the machine as a production cell that can be integrated into larger production lines.

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The functionality and overall structure of a program in an IMM production cell can be mapped to the following main steps that together make up one production cycle: A realistic, non-trivial sequence of actions in a IMM production cell consists primarily of the following steps:

(1) *IMM produces initial, raw part*
(2) *Robot picks raw part from mold*
(3) *Robot places raw part for cooling on intermediary location.* Alternatively, if the part was not correctly produced, the robot puts the part in the garbage.
(4) *Robot takes cooled part.*
(5) *Robot picks solidified part from previous run.* This step includes moving safely towards and into the machinery area and synchronizing with the opening of the mold.
(6) *Robot inserts cooled, raw part into mold.* If it is impossible to insert components while already holding parts, then this step needs to be postponed to the end of a cycle.
(7) *IMM produces added-on final part.* With multiple forms, the IMM may also produce an additional, raw part simultaneously.
(8) *Robot detaches and disposes sprue* after moving out of the machinery.
(9) *Robot places solidified, final part.* A part is placed either (a) at the indicated placement area (e.g. conveyor belt), (b) at the quality inspection site, if an inspection of the part was requested, or (c) in the garbage for recycling, if the part was detected as faulty, otherwise.

The two main use cases of an injection moulding machine are molding parts or molding onto existing parts, each requiring just a single form. In the above list of steps, the first use case consists of steps 1, 2, 8, 9 while the second use case consists of steps 4, 5, 6, 7, 8, and 9. Our scenario combines both these aspects and thus requires an IMM with two forms and hence added complexity. Then the IMM exhibits two forms: one for molding the initial raw part, and the other for molding the add-on onto the raw part to obtain the final part. Depending on the necessary cooling time and molding time, multiple intermediary cooling locations are required for maximizing throughput.

Programming this behavior in detail requires specifying (i) how to bring the cell into a ready production state from an unknown previous shutdown, (ii) transitioning into a cyclic production process, (iii) and handling expected deviations for quality inspection and faulty pieces. The first part requires to consider all different states the production cell might be in when it gets shut down due to production completion, safety stop, program exception, etc. This potentially results in parts remaining in the mold, on the robot gripper, on cooling locations, etc. that need to be removed from the cell or alternatively the production process continued with.

The second part involves the action that are needed (or not yet needed) for a regular cycle. In the simplest case of just molding a part and then placing it on the conveyor, the regular cycle is immediately reached. In a complicated case involving multiple cooling locations, the IMM will mold only raw parts until the locations are all filled up, and only then will the robot be picking a cooled, raw part for add-on molding. Then the cell will reach a cyclic behavior where the IMM molds a raw part (for cooling) and a final part at the same time, and the robot will place a raw part for cooling and retrieve a cooled part for add-on molding in every cycle.

Once the cyclic behavior is reached, the cell behavior remains mostly identical from one cycle to the next with exception to the occasional placement of quality inspection or faulty parts. Typically the goal is to optimize this cyclic behavior for maximum utilization of the IMM, respectively maximizing production throughput of the cell with little opportunity or need to significantly re-plan the cycle upon a deviation (i.e., QA inspection or faulty part).

2.2 Introduction to PDDL

The Planning Domain Definition Language (PDDL) [11] emerged in the symbolic AI community more than two decades ago to describe planning problems in a generally agreed upon format. PDDL consists of two types of artifacts: a domain description and a set of one or more problem description(s). The former describes the *objects* that are known in the domain, the *predicates* over these objects (i.e., the fact that can be known about these object), and *actions* that describe under which preconditions certain effects are obtained. The latter artifacts describe a concrete problem instance using the vocabulary from the domain definition. A problem description thus defines which object instances exist, which predicates over these objects are true and the desired goal.

Listing 1: PDDL Domain description excerpt

```
1  (define (domain imm)
2   (:requirements :strips ...)
3   (:types mold form robot gripper product ...)
4   (:predicates
5      (isAt ?g - robot ?pos - waypoint)
6      (emptyGripper ?g - gripper)
7      (onGripper ?p - product ?g - gripper)
8      (posForPickProd ?pos - waypoint)
9         ...
10  )
11  (:functions
12    (countProdInForm ?m - mold)
13    (prodState ?p - product)
14  )
15  (:action pickRaw
16   :parameters (?g - gripper ?p - product ... )
17   :precondition (and
18                  (not (onGripper ?p ?g))
19                  (emptyGripper ?g)
20                  (isAt ?r ?pos)
21                  (posForPickProd ?pos)
22              )
23   :effect (and
24                  (onGripper ?p ?g)
25                  (not (emptyGripper ?g))
26                  (decrease (countProdInForm ?m) 1)
27                  (assign (prodState ?p) 3)        )))
```

Take the PDDL domain description excerpts in Listing 1. It defines the domain *imm* (line 1), and the required language features (mostly omitted for lack of space, line 2). The available domain object types are *mold, form, robot, gripper, and product* (line 3). The shown set of predicates in lines (5-8) allow to obtain the robot's location, whether (one of its) gripper is empty, whether a particular product is on the gripper, and whether the selected waypoint is suitable for picking the product out of the form, respectively. When the PDDL solver supports *numeric fluents*, it is often more efficient and readable to express predicates as functions (line 12-13). For example, we count how many products are in a mold to be able to ensure that

in a multi-form IMM setup all molded products are picked from the individual forms before the next cycle of molding occurs. We also use functions to model each product's life-cycle state (from a non-existing product, being raw molded, cooling, addon molding, and placed on the conveyor belt) as an integer.

In our example, we define the action *pickRaw* for picking a raw product from one form of the mold. The *parameters* (line 16) define the set of objects that are needed to be checked whether this action can be executed. The example set of preconditions (line 17-22) ensure that the product to be picked is not already on the gripper, that the gripper is empty, that the robot is at a waypoint, and that this waypoint is the correct one for picking the product. The *effect* of this action is then that the product is on the gripper, the gripper is no longer empty, there is one less product in the mold, and that the product's new state is "rawpicked" as represented by '3'.

By default, a planner is time agnostic. All actions occur instantaneous. Modelling actions that require a certain amount of time is possible with PDDL 2.1 *durative-actions* or with PDDL+ *Process*es and *Event*s. These two varants are expressively equivalent, the latter more verbose [4], though.

Listing 2: PDDL Problem description excerpt

```
1  (define (problem example1) (:domain imm)
2   (:objects  prod1 prod2  - product
3              robot1 - robot
4              gripper1 - gripper
5              mold1 - mold
6              form1 - form
7              homePos withMold - waypoint
8   )
9   (:init
10          (emptyGripper gripper1)
11          (posForPickProd withMold)
12          (isAt robot1 homePos)
13          (= (countProdInForm mold1) 0)
14          (= (prodState prod1) 0)
15          (= (prodState prod2) 0)
16  )
17  (:goal (and
18          (= (prodState prod1) 8)
19          (= (prodState prod2) 8)
20  ))
21  (:metric
22     minimize (total-time)     ))
```

One possible problem instance for our example is provided in Listing 2. A problem specification refers to the domain specification (line 1), then defines the available object instances (line 2-8). In our case, we define two products, a single robot, grippers, mold, and form, as well as two robot waypoints. We then initialize the gripper as empty (line 11), and mark the *withMold* waypoint to be the one for picking the raw product (line 12), set the numbers of products in the mold to zero (line 14), and set the products' initial state to zero (lines 15 and 16). Note that in a closed world scenario such as ours, all predicates that are not explicitly set to true are assumed to be false. Next, we define the goals, i.e., the target state of some (or all) of our defined objects. Here we aim to have both products in state "complete" represented here by 8 (lines 19 and 20). When the desired plan should require as little time as possible (e.g., by finding simultaneously executable durative actions) we additionally specify that the the total-time needs to be minimized (line 22).

Domain specification and problem specification are then handed over to a planner. The task of a planner is then to find the sequence of actions brings the system state to the goal state (as defined in the problem definition). Constraints that may not be violated during the plan are either described as part of the problem's goal description (if they are problem specific, such as returning the robot to a specific waypoint at the end) or are already defined as part of the domain if they need to hold for every problem specification (e.g., molding cannot happen when a robot is at the pick waypoint).

Depending on the available actions and their preconditions as well as the size of the problem (i.e., number of object instances that are relevant for obtaining the goal), the planner needs to apply heuristics to intelligently navigate (or prune) the search space. Various approaches are applied that are beyond the scope of this introduction, see [12] for an overview of (PDDL) planners.

2.3 Introduction to HDDL

The Hierarchical Domain Definition Languages was recently proposed to provide a unified representation of hierarchical task network (HTN) planning problems as an extension of PDDL. HTN planning differs from generic planning in the provisioning of domain knowledge for guiding the planner in finding a sequence of actions. The domain knowledge comes in the form of a task decomposition hierarchy: a high-level task is broken down into lower level tasks (iterativly) until a task maps onto an action that can become part of a plan. For each higher-level task there might be one or more alternative lower-level tasks or actions. This form of guidance is useful when its known up-front that certain tasks can only be executed in a particular (partial or total) order. In the IMM domain, for example, a product always needs to be molded first before it can be picked, and only then placed on a conveyor. The concrete plan to carry out this sequence of tasks, however, depends very much on the problem instance and differs especially in complex cells in which form to mold a product in, where to place it for cooling, etc.

HDDL reuses the vocabulary of PDDL and defines the following additional constructs. A *task* defines the parameters that are used as input. A *method* then realized a task by optionally specifying additional parameters, preconditions when the method is applicable, and the order of zero, one, or more subtasks. These subtasks then refer to tasks and/or actions. HDDL comes with constructs for totally ordering the subtasks or to merely define some partial order.

In our example, we take the PDDL domain specification from above and insert after line 27 the HDDL elements provided in Listing 3. The HDDL excerpt in Listing 3 defines a high-level task *t_produce* (line 1) for which a single refining method *m_produce* exists (line 4) . The *m_produce* method specifies that first a task *t_moldRaw* needs to be refined to atomic actions (not shown), then two actions (*pickRaw* and *moveTo*) follow, and then the task *t_placeFinal* needs to be refined (lines 8-13). Note that a method may define more parameters than the task it implements (line 5). The solver then needs to find suitable object instances for these extra parameters based on the method's precondition or any action's preconditions found during refinement. For the latter task *t_placeFinal* two refinement methods *m_placeOnConveyor* (line 20-27) and *m_placeInTrash* exist (line 28-34), each delegating to their respective action (not shown). Which of the two methods the planner selects is guarded by the precondition on the product's state (line 23 and 31, respectively).

Listing 3: HDDL Domain description example extension

```
(:task t_produce
 :parameters (?p - product)
)
(:method m_produce
 :parameters (?p - product ?g - gripper
              ?p1 ?p2 - waypoint ?r - robot )
 :task(t_produce ?p)
 :ordered-subtasks (and
    (t_moldRaw ?p)
    (pickRaw ?p ?g )
    (moveTo ?r ?p1 ?p2)
    (t_placeFinal ?p ?g)
)
(:task t_moldRaw
 :parameters(?p - product)
)
(:task t_place
 :parameters(?p - product ?g - gripper)
)
(:method m_placeOnCoveyor
 :parameters(?p - product ?g - gripper ?pos - waypoint)
 :task(t_placeFinal ?p ?g)
 :precondition (and
    (= (prodState ?p) 3))
 :ordered-subtasks( and
    (placeOnConveyor ?p ?g ?pos)
))
(:method m_placeInTrash
 :parameters(?p - product ?g - gripper ?pos - waypoint)
 :task(t_placeFinal ?p ?g)
 :precondition (and
    (= (prodState ?p) 4))
 :ordered-subtasks( and
    (placeInTrash ?p ?g ?pos)       ))
```

3 CASE STUDY

One of the main factors influencing the modeling of the domain is the set of supported features of the solver. Hence we chose the ENHSP planner [18] for PDDL+ and the PDDL4J-TO planner [13] for HDDL. Given their different feature sets we ended up with three different domain specification variants:

IMMproc utilizes PDDL+ and thus models the duration of molding and cooling actions.

IMMcost avoids duration for performance reasons and utilizes cost functions instead to drive the planner towards more optimal, parallel usage of resources.

IMMhtn reuses the action and predicate definitions from *IMMcost* and adds HDDL constructs for task decomposition but comes without fluents for the product life-cycle state and costs as the PDDL4J-TO planner doesn't support functions.

3.1 Basic IMM Cell Domain Specification

The two most typical use cases consist of molding a new product or to add-on mold onto a raw product, then placing the molded product on a conveyor belt, at a QA inspection station, or in the trash. Our problem domain specification, thus, contains actions to support these two use cases independently.

We decided to keep the IMM domain specification as simple as possible while retaining the complexity in terms of robot and machine interaction, their sequences, and, hence, interleaving of different product life-cycles. We thus chose not to model any triggering of a conveyor belt or stacking products thereon or detailed robot movement paths.

We also abstract from all details of the molding process (such as multi-stage injection) and only track in which of the potentially many mold's forms a product is about to get molded, currently gets molded, or is molded and ready for picking.

Picking and placing is modeled via the robot's gripper, of which there can be more than one. If there are more than one gripper, they are assumed to be usable in parallel (i.e., non-overlapping, able to carry one product per gripper simultaneously). Having multiple robots defined in the problem file is not sensible as we didn't model conflict avoidance and hence more than one robot would be allowed to occupy the same waypoint at the same time.

We summarize the differences of the three domain specification alternatives listed above in Table 1 (top) All three domain modeling variants come with the same set of types and share a significant set of actions. *IMMcost* comes with the shorted specification as the verbose specification of processes and events as found with *IMMproc* is not needed. *IMMhtn* comes with more than double the lines of code compared to *IMMcost*, as the specification of the task hierarchy needs to describe not only the standard sequence of moulding but also skip actions in case a product is left in the production cell from a previous run. The resulting hierarchical task network is available as SOM, displaying tasks for production of a virtual product on the left and an addon molded product on the right with actions used for both "processes" depicted below the dashed line. The eleven grounded actions are in bold font, The 10 tasks and 23 methods are given in italics.

3.2 Extended Domain Specification

We then imitated an engineer adapting the basic domain specification to integrate virtual and addon product molding with an intermediary cooling station along the lines of the motivating scenario in Section 2. In Table 1 (bottom) we list the main differences of *IMMprocR+A*, *IMMcostR+A*, and *IMMhtnR+A* from their basic version, and amongst each other.

For each modeling variant the main driving change is the differentiation between virtual and addon product as reflected in the product life-cycle states, the different grippers applicable in these states, and the distinction between two forms. Specifically in a product's live-cycle we distinguish between having molded a raw product (in the *raw* form) requiring cooling (i.e., *precool*), being *raw* when cooled, and being *addon* when coming out of the second *addon* form. We additionally assume that a product in the former states require a separate gripper (i.e., the *raw gripper*) from a product in the latter *addon* state (i.e., the *addon gripper*) due to having to be picked differently. These changes come with wide reaching impact on action parameters, preconditions, or effects, that otherwise remain semantically the same. For example, the logic for picking and placing of the final *addon* product remains the same, however needs to become aware of picking with the correct gripper from the correct form.

Mold preparing and molding related actions were affected the most as we additionally allow simultaneously molding of raw and addon product, hence also existing actions for individual, separate molding of raw and addon product has to be updated.

Having to track a particular product instance at a specific cooling location came with additional changes. These changes, however,

Table 1: Basis and Extended Domain Specification Comparison

	IMMproc	**IMMcost**	**IMMhtn**
types	product, mold, form, gripper, robot, waypoint	identical	identical
predicates	13: one for robot position, target position, currently moving, locking position for entry, gripper empty, product on gripper, and six for checking positions	9: checks for ongoing of molding, moving, or target position/locking no longer needed	18: same as *IMMcost* plus replacement for the unsupported functions: form state, product quality state, and life-cycle states expressed as 2, 3, and 4 predicates.
functions	6: count of molded products in form, product outcome state, product lifecycle, form state, progress of an PDDL+ process, passed time	5: passed time and progress no longer needed, but cost now tracked	not supported by solver
robot movement	*moveTo* (triggering start), *movingTo* (P) of a robot between waypoints, *arrivedAtPos* (E)	*moveTo* (adapted, instantaneous)	same as *IMMcost*
mold preparing	*pickForInsertion*, *insertRawProductInForm*, *assignVirtualProductToForm*	the same but precondition for current ongoing molding not needed, includes costs	identical to *IMMcost* with functions replaced by predicates
molding	trigger molding with *nextCycle*, *molding* (P), *markVirtualProductForm* (E), *markAddonProductForm* (E), *moldDone* (E), *rawFormFilled* (E), *addonFormFilled* (E),	restructuring into *nextCycleVirtual* and *nextCycleAddon* for instantaneous molding	same as *IMMcost* with functions replaced by predicates
picking	*pickFromRawPos*, *pickFromAddonPos*	same as *IMMproc*	same as *IMMcost* with functions replaced by predicates
placing	*placeInTrash*, *placeForTesting*, *placeOnConveyor*	same as *IMMproc*	same as *IMMcost* with functions replaced by predicates
T/M	n/a	n/a	10 tasks / 23 methods
A/P/E	10 actions / 3 processes / 6 events	11/-/-	11/-/-
LoC	∼300	∼205	∼490
	IMMprocR+A	**IMMcostR+A**	**IMMhtnR+A**
types	additional subtypes of form: *formRaw*, *formAddon*, and subtypes of gripper: *gripperRaw* and *gripperAddon*	same as *IMMprocR+A*	same as *IMMprocR+A*
predicates	16: same as *IMMproc*, additional 2 predicates for product as position mapping, and empty position checking	12: same as *IMMcost* and same additional ones as in *IMMprocR+A*	22: same as *IMMhtn*, same additional ones as in *IMMprocR+A* and additional for cooling state and additional QA state to differentiate between raw and addon product molding result
functions	same as *IMMproc*	same as *IMMcost*	n/a
robot mov.	same as *IMMproc*	same as *IMMcost*	same as *IMMhtn*
mold preparing	adapted to differentiate between grippers, to pick from cooling place only with *gripperRaw*, *insertRawProductAndAssignToForm* same but limited to *gripperRaw* and *formAddon*, *assignVirtualToForm* adapted to use only *formRaw*, (affects parameters only) *pickForInsertion* uses cooling waypoint: precondition and effect extended	similarily adapted to differentiate between grippers and forms, similar adaptations as in *IMMprocR+A*	similarily adapted to differentiate between grippers and forms, similar adaptations as in *IMMprocR+A*
molding	4 events updated: *markRawFormUse/markAddonFormUse* and fillForms limited to their form type; the rest remain identical (affects parameters only)	updated *nextCycleAddon* and *nextCycleVirtual* to different product lifecycle state in precondition and effect, differentiating between form and gripper types, adding of new action *nextCycleBoth* to allow simultaneous molding of virtual and addon product with lower cost.	same adaptations and new action as in *IMMcostR+A* but with function/fluent usage replaced by predicates, no usage of cost
picking	adapted to be limited to their respective gripper and form types, *pickRaw* needs updated product state to signal ready for cooling,	same adaptation as for *IMMprocR+A*	same adaptation as for *IMMprocR+A*, but using predicates instead of functions for product lifecycle and QA state
cooling	new trigger action *placeForCooling*, process, and completion event	new action *placeForCooling*, with immediate cooling effect very similar to *IMMprocR+A*	same new actions as in *IMMcostR+A*, but using predicates instead of functions for product lifecycle and QA state
placing	duplicated action for placing in trash to differentiate between raw and addon product	same duplication for placing in trash as in *IMMprocR+A*	same duplication for placing in trash as in *IMMprocR+A*
T/M	n/a	n/a	12 tasks; 29 methods
A/P/E	12 actions / 4 processes / 7 events	14/-/-	14/-/-
LoC	∼365	∼280	∼660
new LOC	∼65	∼70	∼170
diff LOC	∼15	∼20	∼80

are very local, i.e., only affect the existing fetching of a raw product. Model elements for moving the robot around the production cell required no changes at all.

Inspecting Table 1 in more detail, we notice, that the same type and extent of changes have to be made from the basic to the extended model version for every modelling variant. While the process- and cost-based variants require about the same amount of new lines of code (and have around the same amount of changed lines of code), the hierarchical task network-based variant requires more than twice as many new lines of code and four times as many changed LoC. On root cause of this many changes is the encoding of life-cycle states as predicates and not as a fluents. And second, the hierarchical task networks needs to be adapted to contain additional domain know-how how the two stages (i.e., raw and addon molding) can be combined, and under which circumstances certain tasks can be skipped.

3.3 Lessons Learned

Several planning challenges and tutorials have a domain where one or multiple robots, rovers, trucks, or other objects need to move between multiple locations, positions, or places in an optimal or resource-constraint manner. Hence the planner has to decide which

path and which sequence of locations to take. While also a robot in an IMM cell has to move between different positions for picking and placing, we decided against modeling the various movements paths (i.e., the intermediary positions) in detail as this results merely in a larger search space for the solver (and thus noticeable increased search time) without resulting in better solutions. Typically the positions to reach are determined by the transition of a product state, e.g., from being in the mold, to being on the conveyor with few alternative positions where these actions can be executed, and few, if any, alternative movement paths. The consequence of such simplification is that any plan will contain only the endpoints of a robot movement, and thus a dedicated subsystem needs to translate these endpoints into a precise, detailed path via potentially multiple intermediary positions.

Note that this modeling simplification works under the assumption that a only a single robot operates within the cell, or multiple robots that have no overlapping operation range, or robots that when they occupy the same area will block the overlapping area only for an insignificantly short amount of time (where "insignificant" implies that the maximum amount of time one robot has to wait for the other to move away will not affect its execution plan). As soon as the movement of one robot affects the ability of the other to carry out its task within the allocated time, then their interactions (i.e., their movement paths and locking/reserving of path segments or areas) needs to be explicitly modeled.

4 PERFORMANCE COMPARISON

In addition to a qualitative assessment of the different domain modeling approaches, in this section, we analyse the differences in the solvers' runtime performance. We defined five problems (containing one to five products to go through the molding process) that represent different starting configurations for the simple production scenario ($P1_{base}$ to $P5_{base}$) as well as the extended production scenario ($P1_{ext}$ to $P5_{ext}$). The simple scenario includes one form and one gripper and supports molding of a product in one go, or addon molding onto a raw product. The extended scenario comes with two forms, two grippers, four cooling places, and supports simultaneous molding of a raw product, and addon molding once it has cooled down. The form can also be used to mold either only the raw product, or only the added-on, final product separately. P1 and P2 represent a regular production run, for P3 to P5, we specify left over products from a previous run that may still reside in the mold form and/or on the gripper. As the extended production cell also for much more variety to have products left over from a previous run, we created additional problem files ($P6_{ext}$ to $P7_{ext}$) containing six and seven products, respectively. In $P7_{ext}$, we have one product on either gripper, a product in either form, a product at one of the cooling locations, and aim to mold two regular products, one that goes to the quality inspection station, and one to be placed on the conveyor. In the scenario, the five products from the previous round have to be discarded, rather than continuing with their production process. The used domain specifications and problem files are available as part of the supporting online material [1].

We adapted each problem specification across the three modeling variants to be as close as possible (while being semantically identical) to enable a fair comparison. For the cost-based model

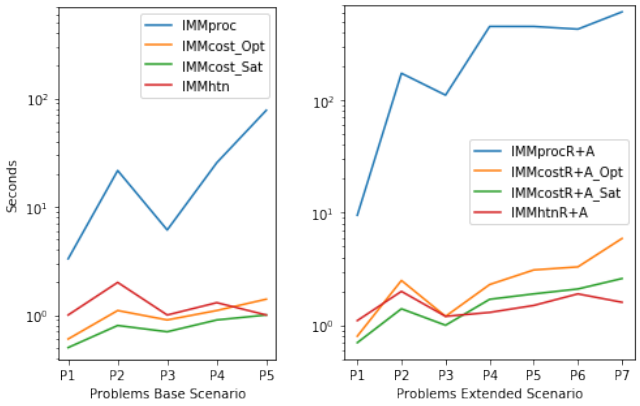

Figure 1: Solver Performance: average runtime (in seconds).

variant we additionally compare the runtime for finding any solution (IMMproc_Sat), i.e., a satisfiable solution, and an optimal solution (IMMproc_Opt) for a total of four solver configurations. We collected average runtime measurements for each problem instance and solver configuration pair on a standard laptop, using the solvers' default configuration over five iterations. Figure 1 depicts the average runtime in seconds for each solver configuration and problem instance (note the graphs' log-scale y-axis).

We observe that for the simple scenario there is not much relevant difference among the time-unaware models with the runtime remaining between 0.5 and 1.4 seconds, regardless of problem instance. In contrast the time-aware model quickly grows to beyond several seconds to over a minute for $P5_{base}$. The extended scenario show a similar behavior with little difference for $P1_{ext}$ to $P5_{ext}$ but clearer distinction beyond that. As the complexity of the problem increases in $P6_{ext}$ and $P7_{ext}$, we see that finding a cost optimal solution to require more time, followed by the cost satisfiable one, and the hierarchical task network-based approach being the fastest. The time-aware model exceeds six minutes of runtime with $P4_{ext}$ and even ten minutes with $P7_{ext}$.

In summary, obtaining time-aware plans requires one to two orders of magnitude more time than time-unaware plans.

5 DISCUSSION AND IMPLICATIONS

Towards flexible production system programming, the main challenge is not just finding optimal production sequences but the more difficult task of enabling the extension of standard behavior to support novel use cases not foreseen by the manufacturer.

Answering RQ1 on whether PDDL or HDDL (and respective solvers) are suitable for finding optimal production sequences, we conclude that, in general, it's a feasible approach but far from practical applicability for the following reasons.

One needs to keep track of individual product instances in a PDDL/HDDL plan, when each of them is in a different production state but present in a production cell at the same time. The consequence of having multiple product instances is the difficulty of turning a sequential plan (as provided by the solver) into a cyclic one as an IMM typically produces not one but many identical products.

Especially difficult is obtaining a transition from an initialization phase with various amounts of cleanup actions, into a cyclic plan with hardly any unexpected deviations. Even in the case of a minor, expected deviation such as having to place a product onto the quality inspection site would invalidate a plan, subsequently requiring re-planing all the while continuing with the production. Planning duration, however, is not at the time scales needed for production yet. From the performance evaluation, we learn that time-unaware planning is reasonably quick, in a real production environment, however, time efficient plans are absolutely necessary.

An interesting mitigating approach here could be to derive upfront multiple start and intermediary conditions and generate a set of plans. Process mining approaches from the BPM community could then perhaps extract a unifying process model that accounts for most of the possible deviations, and then this process model controls the production cell.

We observed an additional limiting behavior of HDDL: the order of tasks to solve influenced the solving duration and sometimes even the ability to generate a plan at all. Hence, additional care is required to formulate the problem instance.

Answering RQ2 on the extent of changing domain specifications to support new use cases, we come to the conclusion that additional research and engineering support is needed to support domain experts that are typically not PDDL or HDDL experts. As we have observed over the previous pages, supporting new use cases is not just a matter of dropping in additional logic at one place and keeping everything else the same. In our real-world inspired use case, the necessary cascading changes affected large parts of the overall domain specification (regardless of the modeling variant), here due to product a different life-cycle as well as distinction between form subtypes and gripper subtypes. Understanding all the affected locations in the domain definition and ensuring that any changes are correctly made is non-trivial. Given the current limitations in engineering support (see Section 6), we hypothesize, that this procedure is too complicated for the typical domain engineer to conduct in a timely and satisfactory manner.Especially the HDDL-based approach requires a detailed understanding what the state of a product and production cell is in after each task: relevant to correctly define skip actions that become necessary when products remain in the cell from the previous cycle.

One future way to mitigate this aspect could be restricting the engineers' ability to modify the complete domain specification to clearly identified extension points. For example, in our case, modeling a products life-cycle stats and their transitions as a UML state chart and from this model then derive predicates. Exemplary, complementary research directions could be investigations into algorithms that detect the impact of changes across PDDL/HDDL actions, or algorithms that detect logical inconsistencies among preconditions due to adaptations. Deriving of partial plans with explanations where of why a solver found no overall plan would be equally helpful to fixing inconsistent domain specifications or determining unsolvable problem instances.

Even as HDDL comes with additional specification overhead, it has more potential to become industry relevant than pure PDDL. The primary reason is in it's ability to include domain specific know-how and thus more efficiently derive plans. Moreover, restricting the possible behavior is typically an advantage in industrial settings to obtain more predictable behavior compared to a pure PDDL-based domain specification. In the latter case a planner might come up with surprising, potentially dangerous, behavior for a new, previously unseen problem setting. This quickly becomes a safety relevant aspect when humans participate in a production cell.

6 RELATED WORK

There have been some efforts over the past two decades to apply PDDL in industrial settings to various degrees of success. Wally et al. [22] describe one possible way to represent IEC 62264 models in PDDL and evaluate it for an assembly process scenario involving autonomous shuttles and robots. Their primary focus is on how to map IEC 62264 models in general to PDDL. They similarly experienced performance problems when considering temporal aspects. They utilized the LPG-td PDDL solver [6], that supports durative actions to this end. Huckaby et al. [8] use SysML to model system capabilities and process specification for subsequent transformation into PDDL. They evaluated their approach by generating plans for coordinating multiple robots in mounting a single door onto a car. Their approach remains unaware of action duration. No performance discussion on plan generation is available. Rimani et al. [15] also describe a SysML-based approach for manually generating HDDL specifications. In constrast, Rogalla et al.[16] propose directly encoding production and planning know-how in PDDL. Their evaluation scenario consisted of a rotary transport system to which four production stations are connected. Optimizing the makespan of a set of 10 orders or more didn't yield an optimal plan within 30min of search time.

Hoebert et al. [7] propose a mapping from the Web Ontology Language (OWL) to PDDL for speeding up the reconfiguration of robot movements and applied it to a pick and place scenario. Performance information on plan generation was provided only for individual product picking sequences with no focus on optimizing multiple production instances within a cell. Kootbally et al. [10] similarly use a semantic description from which PDDL domain and problem files are generated to control a robot preparing a parts that are required for the assembly of a single product.

Bolender et al. [3] utilize PDDL as a fallback mechanism for self-adaptation of an injection molding machine, when their primary Case-Based Reasoning approach provide no suitable adaptation plan due to lack of similar situations in the part. Their work differs from ours in two key aspects: first, the problem focuses on configuring molding parameters for the next cycle and not how to optimally sequence multiple production cycles with parallel behavior as in our case. Second, their problem specification is fixed and there is no need to adapt it, whereas we investigate what the implications are on extending the specification to include additional behavior. Setta et al. [19] applied planing with PDDL to optimally schedule oil tanker arrival and unloading at a refinery dock. Their experiments, even while not considering temporal aspects, hit memory and runtime limits when trying to optimize a plan. Fritz [5] investigated the use of PDDL for deriving the sequence of commands for a CNC machine and found it unusable due to lack of representing geometric properties and operations.

Some work aims to overcome the limitations of PDDL to adequately handle uncertainty and repetition on the shopfloor. Rogalla

and Niggemann [17] describe an approach to deal with the non-determinism of production, e.g., the uncertainty what a sensor reading will be or to obtain runtime information from the shopfloor. The authors propose to solve subproblems encoded in PDDL first that mitigate this uncertainty and then apply regular planning. Yet, their application is limited to production of single products, unable to model cyclic production or coordination of parallel production steps involved. Asai and Fkunaga [2] describe a technique for deriving cyclic plans. The limitations are that only identical products can be produced, in the exact same order, and multiple instances of the production plan cannot interleave. Hence a new cycle with the next product can only start upon finishing the prior product.

Engineering of PDDL domain and problem files has received some attention with support for integrated development environments (IDEs) appearing over the past years. Strobel and Kirsch [20] compared existing engineering support such as PDDL Studio [14], itSimple[21], PDDL-mode[1], Planning.domains[2], and vscode-PDDL[3] before introducing their MyPDDL IDE. For the purpose of this paper, we have used VSCode PDDL, which is a plugin for Visual Studio Code. All these engineering support tools focus primarily on writing syntactically correct domain and problem files. A major short coming is in debugging and testing of domain specifications, or understanding the impact of changes when evolving the domain specification. To the best of our knowledge, there is currently no support for determining whether a solver fails to find a plan because the domain specification contains a logical error, the problem specification describes an unsolvable problem (while assuming the domain to be correct), or whether the problem is just too complex for the solver to find a solution in an acceptable timeframe.

7 CONCLUSIONS

In this paper we analysed to what extent a non-programmer would have to adapt a domain specification to evolve a standard injection molding production cell to a complex, realistic scenario. We identified three main reasons why PDDL/HDDL are not yet practically applicable for this purpose: performance of solvers, lack of cyclic production plans, and most importantly for domain experts: insufficient engineering support during the evolution task itself. Yet, we expect that HDDL/PDDL will play a major part in production control, first in highly adaptive environments where true batch size one production needs continuous replanning anyway. Here HDDL might be more practically relevant than PDDL as it enables to integrate more domain know-how for planning, thereby reducing time-to-plan and obtaining more predictable plans.

ACKNOWLEDGMENTS

The research reported in this paper has been funded by BMK, BMDW, and the State of Upper Austria in the frame of SCCH, part of the COMET Programme managed by FFG.

REFERENCES

[1] [Authors.]. Supporting Online Material. https://figshare.com/s/8315f52edb597fb7836a. Accessed: 2021-12-15.

[2] Masataro Asai and Alex Fukunaga. 2014. Fully Automated Cyclic Planning for Large-Scale Manufacturing Domains. *Proceedings of the International Conference on Automated Planning and Scheduling* 24, 1 (May 2014), 20–28.

[3] T. Bolender, G. Burvenich, M. Dalibor, B. Rumpe, and A. Wortmann. 2021. Self-Adaptive Manufacturing with Digital Twins. In *2021 2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) (SEAMS)*. IEEE Computer Society, Los Alamitos, CA, USA, 156–166.

[4] Maria Fox and Derek Long. 2006. Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research* 27 (2006), 235–297.

[5] Christian Fritz. 2016. Automated Process Planning for CNC Machining. *AI Magazine* 37, 3 (2016).

[6] Alfonso Gerevini, Alessandro Saetti, and Ivan Serina. 2006. An Approach to Temporal Planning and Scheduling in Domains with Predictable Exogenous Events. *J. Artif. Int. Res.* 25, 1 (feb 2006), 187–231.

[7] Timon Hoebert, Wilfried Lepuschitz, Markus Vincze, and Munir Merdan. 2021. Knowledge-driven framework for industrial robotic systems. *Journal of Intelligent Manufacturing* (2021), 1–18.

[8] Jacob Huckaby, Stavros Vassos, and Henrik I. Christensen. 2013. Planning with a task modeling framework in manufacturing robotics. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 5787–5794.

[9] Daniel Höller, Gregor Behnke, Pascal Bercher, Susanne Biundo, Humbert Fiorino, Damien Pellier, and Ron Alford. 2020. HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 06 (Apr. 2020), 9883–9891.

[10] Zeid Kootbally, Craig Schlenoff, Christopher Lawler, Thomas Kramer, and Satyandra K Gupta. 2015. Towards robust assembly with knowledge representation for the planning domain definition language (PDDL). *Robotics and Computer-Integrated Manufacturing* 33 (2015), 42–55.

[11] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. 1998. *PDDL—The Planning Domain Definition Language.* Technical Report CVC TR98003/DCS TR1165. Yale Center for Computational Vision and Control, New Haven, CT.

[12] Tim Niemueller, Till Hofmann, and Gerhard Lakemeyer. 2018. CLIPS-based execution for PDDL planners. In *ICAPS Workshop on Integrated Planning, Acting and Execution (IntEx)*.

[13] Damien Pellier and Humbert Fiorino. 2021. Totally and Partially Ordered Hierarchical Planners in PDDL4J Library. In *Proceedings of 10th International Planning Competition: Planner and Domain Abstracts – Hierarchical Task Network (HTN) Planning Track (IPC 2020)*. 17–18.

[14] Tomas Plch, Miroslav Chomut, Cyril Brom, and Roman Barták. 2012. Inspect, edit and debug PDDL documents: Simply and efficiently with PDDL studio. *System Demonstrations and Exhibits at ICAPS* (2012), 15–18.

[15] Jasmine Rimani, Charles Lesire, Stéphanie Lizy-Destrez, and Nicole Viola. 2021. Application of MBSE to model Hierarchical AI Planning problems in HDDL. In *2021 Knowledge Engineering for Planning and Scheduling Workshop at ICAPS'21*.

[16] Antje Rogalla, Alexander Fay, and Oliver Niggemann. 2018. Improved Domain Modeling for Realistic Automated Planning and Scheduling in Discrete Manufacturing. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, Vol. 1. 464–471.

[17] Antje Rogalla and Oliver Niggemann. 2017. Automated process planning for cyber-physical production systems. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 1–8.

[18] Enrico Scala, Patrik Haslum, Sylvie Thiebaux, and Miquel Ramirez. 2016. Interval-Based Relaxation for General Numeric Planning. In *Proceedings of the Twenty-Second European Conference on Artificial Intelligence* (The Hague, The Netherlands) *(ECAI'16)*. IOS Press, NLD, 655–663.

[19] Fernando Moreira Sette, Tiago Stegun Vaquero, Song Won Park, and Jose Reinaldo Silva. 2008. Are Automated Planners up to Solve Real Problems? *IFAC Proceedings Volumes* 41, 2 (2008), 15817–15824.

[20] Volker Strobel and Alexandra Kirsch. 2020. *MyPDDL: Tools for Efficiently Creating PDDL Domains and Problems.* Springer International Publishing, Cham, 67–90.

[21] Tiago Stegun Vaquero, Flavio Tonidandel, and José Reinaldo Silva. 2005. The itSIMPLE tool for modeling planning domains. *Proceedings of the First Intl. Competition on Knowledge Engineering for AI Planning, Monterey, California, USA* (2005).

[22] Bernhard Wally, Jiří Vyskočil, Petr Novák, Christian Huemer, Radek Šindelář, Petr Kadera, Alexandra Mazak, and Manuel Wimmer. 2019. Production Planning with IEC 62264 and PDDL. In *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, Vol. 1. 492–499.

[1] http://rakaposhi.eas.asu.edu/planning-list-mailarchive/msg00085.html
[2] http://planning.domains/
[3] https://github.com/jan-dolejsi/vscode-pddl