CLIPS-based Execution for PDDL Planners

Tim Niemueller and **Till Hofmann** and **Gerhard Lakemeyer**
Knowledge-Based Systems Group,
RWTH Aachen University, Germany

Abstract

Integrating planning and execution which treats either component as a black box may lead to disparate representations of the domain or information currently known. Consistency and bidirectional information flow are then hard to ensure. However, the separation of these concerns is still useful from an integration point of view.

In this paper, we discuss the integration of planning systems using the Planning Domain Definition Language (PDDL) with an executive based on the CLIPS rule-based production system. In particular, we describe how we achieved one common and unified domain model used by both systems and some additions we add for the execution model. We also show how the execution model enables effective execution monitoring and selective replanning.

1 Introduction

Agents and robots that perform in dynamic environments need to reason about their course of action to achieve their goals. On the task-level, this requires a system combining planning and execution.[1] Planning is the process of determining actions (and their ordering – total or partial – and necessary intermediate conditions). The outcome of this process is then passed on to execution, which interprets this plan and invokes and monitors actions that effect the necessary change to achieve specific goals.

There are a wide variety of systems integrating both, planning and execution. Often, these systems are in some way biased about which component constitutes the top-most authority, i.e., the part of the system which takes or generates goals and controls the reasoning and execution process. Sometimes, a planner is the top-level system and execution is mere action dispatching downstream, with errors triggering another planning run. The other view can be that the executive uses the planner as a black box which is called at suitable times.

In this paper, we propose a formulation for the integration of planning systems using the Planning Domain Definition Language (PDDL) an executive based on the CLIPS rule-based production system as part of an on-going effort towards a CLIPS Executive (CX). While taking the standpoint of having the executive as top-most controller, we use a common *domain model* including available operators, predicates, and known facts. We focus on the STRIPS fragment of PDDL with types. We describe a CLIPS representation of an *execution model* that is directly derived from the exact same PDDL domain file used by the planning system. It features some extensions made for plan execution, for example to describe sensed predicates. Effects on such predicates can be observed and should therefore not be applied directly; instead, the executive should wait for the effect to occur. This can be useful to model processes that are triggered by the agent but that do not cause an immediate effect, or for exogenous actions which are not under the control of the agent itself. However, since most PDDL models do not account for these kinds of actions explicitly,[2] the planning model must assume these actions to have deterministic and immediate effects. During execution, we merely observe sensed predicates and use deviations as input for execution monitoring. The *planner model* contains the subset of information known about the environment suitable for consumption by a planner. An automatically synchronized *world model*, a superset of the planner model, contains all information relevant to the CX.

In the following, we discuss some related work in Section 2 and provide an architecture overview in Section 3. Our PDDL representation in CLIPS is presented in Section 4, the PDDL planner integration in Section 5. We detail plan execution and monitoring in Section 6, before we conclude.

2 Related Work and Background

Numerous systems have been proposed in the past that handle task execution for autonomous systems. A task describes a concrete and executable specification that aims to achieve or maintain some goal through the execution of such as a plan, a policy, or a program. Often a task makes use of primitive actions as a means to effect change in the environment.[3]

[1]Many systems, for example in experimental robotics, often forgo a lookahead planning system and rather perform simple action selection or pursue fixed plans or scripts.

[2]PDDL+ (Fox and Long 2006) supports events to model externally triggered changes. However, we do not intend to account for these effects at planning time, where this can be tedious and costly, but at execution time where we can cope with contingencies easier.

[3]Some formalisms simply see an action as a basic task.

Planning and execution systems greatly vary in terms of the language or programming interface used for task and action specification.

2.1 Executives

In the following, we focus on execution systems and how they integrate (with) planning systems.

PLEXIL The Plan Execution Interchange Language (PLEXIL) (Verma et al. 2006) is a representation language for plans in automation. The PLEXIL Executive is an implementation to interpret and execute PLEXIL plans. A plan is decomposed into a set of typed nodes which serve a specific function, such as making an assignment or issuing a command to the controlled system. PLEXIL supports concurrency, program flow primitives (conditionals, loops), and explicit sensing of external information. The executive deals with plan execution only. It does not invoke or control a planning process. However, a prototype has been developed to externally combine a planner with PLEXIL (Muñoz, R-Moreno, and Castaño 2010).

ROSPlan ROSPlan (Cashmore et al. 2015) is a framework for task planning that describes a number of exchangeable components and a set of message types to interconnect these. Such components are, e.g., planner integration (problem generation, invocation, result parsing), and fact base storage. The execution system is rather basic and hence called plan dispatcher. After a plan has been generated, the dispatcher publishes messages for each actions (one-by-one) which must be interpreted and achieved through external programs. Even though ROSPlan's default planner POPF produces temporal plans with concurrency, the current internal representation only yields sequential execution.[4] The dispatcher does not evaluate preconditions of actions during execution and hence may invoke actions which cannot be accomplished. Effects of actions are not automatically applied to the fact base, but the external action provider must do this.

CLIPS Agent The rule-based production system CLIPS provides the basis for an incremental task-level reasoning system (Niemueller, Lakemeyer, and Ferrein 2013). It does not provide an explicit task specification language. Rather, the behavior is defined in a knowledge-based reactive fashion, where situation classifiers directly decide on the next action to perform whenever the agent is currently idle. Actions are modeled as external functions and monitoring is performed through rules observing updates to the fact base. The system does not perform any planner integration.

CLIPS SMT A later revision of the aforementioned system was extended to integrate with an SMT-based planning system (Niemueller et al. 2017), that featured optimization through on-line constraint adaptation. It featured an explicit multi-actor plan representation that served as an interface to separate the planner from the execution. Once a plan is generated, macro operations from the plan are replaced by the respective sequence of actions. Then, action selection does not occur based on a situation classification as with the CLIPS Agent, but rather based on the expanded plan. A shortcoming of the actor-based plan representation is the need for synchronization constructs to model that some plans may not progress until certain points have been reached in other plans.

[4]Yet unreleased code in the development branch of ROS-Plan (`https://github.com/KCL-Planning/ROSPlan`) seems to improve this. However, we could not verify this in time.

OpenPRS The Procedural Reasoning System (PRS) is a high-level control and supervision framework to represent and execute plans and procedures in dynamic environments (Ingrand et al. 1996).[5] PRS has three main elements: a *database* containing facts representing the belief about the world, a *library of plans* (or procedures) that describe a particular sequence or policy to achieve a certain (sub-)goal, and a *task graph* which is a dynamic set of tasks currently executing (Niemueller et al. 2016). Tasks are specified in terms of small programs (supporting loops, conditionals, and recursion), called OPs. OPs have logic formulas as activation conditions, that, if matched, invoke an OPs. A specialty is that multiple OPs can be executed in parallel. However, this can make proper plan design non-trivial since conditions such as race conditions must be handled. OpenPRS does not directly support planner integration, but typically OPs (partial plans) are written (or graphically designed) manually.

ActorSim ActorSim (Roberts et al. 2016) is an implementation of Goal-Task-Networks (GTN). Goals are considered as a top-level construct that have a specific life cycle. Once a goal is selected, it is expanded, which may invoke an external planner. However, ActorSim itself does not provide a ready-to-use integration for planning systems. Expansion generates a task, which is then executed through invoking actions on the controlled system.

ASP-TBP A recent approach utilizes an extension of the CLIPS Agent as an executive for time-bounded planning using Answer Set Programming (ASP) (Schaepers et al. 2018). It generates plans with a time-limited lookahead (typically up to 3 minutes) that already contains actor assignments for each sub-task. The planner runs virtually continuously concurrent to execution. Whenever a new and better (according to some metric) plan is found that is compatible with the current execution state, the new plan is published through a globally shared database. A simplified executive on the executing agents then retrieves new tasks from this database when it becomes idle. The plan does contain expected task durations allowing for reporting delays.

Kirk/RMPL The Reactive Model-based Programming Language (RMPL) (Williams et al. 2003) provides the means to describe rich control programs including loops and conditionals. It also supports preemption of programs by specifying necessary conditions during the execution of some (partial) program. Furthermore, it supports concurrency and non-deterministic choice. RMPL is amenable reactive planning. Kirk is an RMPL-based planner/executive (Kim, Williams, and Abramson 2001) that transform an RMPL specification in a temporal plan network for inter-

[5]OpenPRS is the most widely available PRS version.

leaved planning and execution. There are no openly available implementations for RMPL or Kirk, which we therefore could not evaluate first-hand.

GOLOG GOLOG (Levesque et al. 1997) is a high-level programming language based on the Situation Calculus (McCarthy 1963; Reiter 2001). Similar to RMPL, GOLOG allows loops and conditionals, and also supports non-deterministic choice. GOLOG has been extended for interleaved concurrency (De Giacomo, Lespérance, and Levesque 2000), on-line execution (De Giacomo, Lespérance, and Levesque 2000), and execution monitoring (De Giacomo, Reiter, and Soutchanski 1998). GOLOG can also use PDDL for planning (Claßen et al. 2012) with an `achieve` operator that delegates search to a PDDL planner, and continual planning (Hofmann et al. 2016), which interleaves planning with plan execution, plans for acquiring missing knowledge, and monitors the environment for unexpected events and exogenous actions.

2.2 CLIPS Rule-Based Production System

CLIPS (Wygant 1989) is a rule-based production system using forward chaining inference based on the Rete algorithm (Forgy 1982) consisting of three building blocks (Giarratano 2007): a fact base or working memory, the knowledge base, and an inference engine. *Facts* are basic forms representing pieces of information in the fact base. They usually adhere to structured types. The *knowledge base* comprises heuristic knowledge in the form of rules, and procedural knowledge in the form of functions. *Rules* are a core part of the production system. They are composed of an antecedent and consequent. The antecedent is a set of conditions, typically patterns which are a set of restrictions that determine which facts satisfy the condition. If all conditions are satisfied based on the existence, non-existence, or content of facts in the fact base the rule is activated and added to the agenda. The consequent is a series of actions which are executed for the currently selected rule on the agenda, for example to modify the fact base. *Functions* carry procedural knowledge and may have side effects. They can also be implemented in C++. In our framework, we use them to utilize the underlying robot software, for instance to communicate with the reactive behavior layer described below. CLIPS' *inference engine* combines working memory and knowledge base performing fact updates, rule activation, and agenda execution until stability is reached and no more rules are activated. Modifications of the fact base are evaluated if they activate (or deactivate) rules from the knowledge base. Activated rules are put onto the agenda. As there might be multiple active rules at a time, a conflict resolution strategy is required to decide which rule's actions to execute first. In our case, we order rules by their salience, a numeric value where higher value means higher priority. If rules with the same salience are active at a time, they are executed in the order of their activation (Niemueller et al. 2016).

3 System Architecture and Models

The CLIPS Executive is integrated using the Fawkes robot software framework. It consists of several components, such as the CLIPS run-time environment, a PDDL-to-CLIPS parser, a planner integration component, and a reactive behavior component.

Fawkes (Niemueller et al. 2010) is a component-based software framework with a blackboard communication architecture. It provides the basic building blocks for the integrated system. The CLIPS environment and the planner component communicate through a robot memory based on the MongoDB-driven robot database (Niemueller, Lakemeyer, and Srinivasa 2012). The basic behaviors are provided through the Lua-based Behavior Engine (Niemueller, Ferrein, and Lakemeyer 2009). It provides a development and execution environment for skills modeled as hybrid state machines and accessible through execution functions. Skills can be structured hierarchically to enable building more complex actions (which are still reactive and can only perform local choices).

3.1 Models

In the following, a number of different models with varying scopes are necessary for the description of the PDDL integration. We briefly introduce each of these models.

Domain Model D The domain model is akin to a PDDL domain and contains descriptions of operators (action templates), predicates, and object types. One of the most important aspects is that both, the planner and the CX, use the same domain model.

Planner Model P The planner model contains the facts and object instances the planner can represent and reason about. In the case of PDDL, this is the set of initial facts and objects that will be stored in the problem file. For this paper, we assume a symbolic model making the closed world assumption.

Execution Model E The execution model is a superset of (and thus extended) domain model. It may contain enriched operator descriptions (for example mentioning effects only relevant during execution) and designate sensed predicates (cf. Sections 4 and 6).

World Model W The world model contains all relevant information known about the internal and external environment. It is a superset of P; in addition to the facts needed by the planner, it contains facts that are irrelevant for planning but used during execution, e.g., precise positions and information about other robots. It features a richer representation supporting lists, numbers, symbols, and strings. Facts in the world model are identified by a unique key. The world model is the only interface to ingest information into the executive (aside from action feedback).

In short, E extends D with additional operators and operator properties, while W extends P by additional facts needed for execution. In other words, P is the restriction of W to facts and objects required for planning. Models P and W are synchronized automatically, that is, any update in W is reflected in P, and vice versa. The planner does not modify P directly, rather, it uses it to formulate the planning problem. Both D and E are generated from the PDDL domain description, and additional properties in E are asserted by the domain designer.

```
1 (at ?r -robot ?m -location ?side -side)
2 (at R-1 C-BS INPUT)
```

Listing 1: PDDL predicate declaration and instance.

```
1  (deftemplate domain-predicate
2    (slot name (type SYMBOL)
3      (default ?NONE))
4    (slot sensed (type SYMBOL)
5      (allowed-values FALSE TRUE))
6    (multislot param-names (type SYMBOL))
7    (multislot param-types (type SYMBOL))
8  )
9
10 (domain-predicate
11   (name at) (sensed FALSE)
12   (param-names r m side)
13   (param-types robot location side)
14 )
```

Listing 2: CLIPS template and instance for predicates.

4 PDDL Domain Representation in CLIPS

Planning and execution are based on a common *domain model*. The CX *execution model* is derived directly from this domain model and is specified as a PDDL domain. A dedicated parser reads the domain file and asserts the necessary structures in CLIPS. In the following, we give an overview of these structures.

Predicates Predicates carry information about the world. PDDL uses a symbolic representation. An example representing a robot's position is shown in Listing 1 (l. 1). This is translated into a CLIPS fact using the template in Listing 2 (ll. 1–8). The name, which may not be empty, as indicated by the special **?NONE** default value, is simply the head of the PDDL predicate. The multislot `param-names` is the list of parameter names defined in the PDDL predicate. The multislot `param-types` is the list of the respective parameter types. An example for the representation of the PDDL `at` predicate is shown in lines 10–14.

The slot `sensed` is an extension for the execution model E. If set to `TRUE`, it indicates that this is a predicate under exogenous control, i.e., it is not directly influenced by the agent but rather update from an external entity. Therefore, the value of a sensed predicate is not changed when applying the effects of an action (cf. Sensed Effects in Section 6).

```
1 (deftemplate domain-fact
2   (slot name (type SYMBOL)
3     (default ?NONE))
4   (multislot param-values)
5 )
6
7 (domain-fact (name at)
8  (param-values R-1 C-BS INPUT)
9 )
```

Listing 3: CLIPS template and instance for facts.

```
1  (deftemplate domain-precondition
2   (slot name (type SYMBOL)
3    (default-dynamic (gensym*)))
4   (slot part-of (type SYMBOL))
5   (slot type (type SYMBOL)
6    (allowed-values conjunction negation))
7   (slot grounded (type SYMBOL)
8    (allowed-values FALSE TRUE))
9   (slot grounded-with (type INTEGER))
10  (slot is-satisfied (type SYMBOL)
11   (allowed-values FALSE TRUE))
12 )
13 (deftemplate domain-atomic-precondition
14  (slot name (type SYMBOL)
15   (default-dynamic (gensym*)))
16  (slot part-of (type SYMBOL))
17  (slot predicate (type SYMBOL))
18  (multislot param-names (type SYMBOL))
19  (multislot param-constants)
20  (multislot param-values)
21  (slot grounded (type SYMBOL)
22   (allowed-values FALSE TRUE))
23  (slot grounded-with (type INTEGER))
24  (slot is-satisfied (type SYMBOL)
25   (allowed-values FALSE TRUE))
26 )
```

Listing 4: CLIPS templates for operator preconditions.

A ground instance of a predicate (e.g., an initial fact) is represented as **domain-fact** (Listing 3, ll. 1–5). An instance stating that robot `R-1` is at the `INPUT` side of the machine `C-BS` is shown in lines 7–9 (akin to Listing 1, line 2).

Actions An action is represented by a number of templates, one for the operator name and parameters, and several for the action's *precondition* and its *effects*. As CLIPS does not support nested templates, the precondition of an action is split into several facts. A **domain-precondition** is a non-atomic precondition, i.e., a conjunction or a negation, with sub-conditions. A **domain-atomic-precondition** is an atomic precondition and always refers to a specific predicate. The PDDL precondition is decomposed into a tree of preconditions. The root is always non-atomic, typically a conjunction. Atomic preconditions can only be a child of compound preconditions. Leaves which are atomic preconditions represent the respective predicate requirement. Conjunctive leaves are considered to be `TRUE`, negation leaves evaluate to `FALSE`. Additionally, a **domain-precondition** can also be part of another **domain-precondition**, which allows nested preconditions. The templates of the preconditions are shown in Listing 4. A **domain-precondition** has a name, which is automatically set to a unique name if no name is given. The name is used to specify the precondition as a parent condition of another precondition, which is specified with the slot `part-of`. A non-atomic precondition can be of type `conjunction` or `negation`. A **domain-atomic-precondition** always refers to a `predicate` and has parameter names and constants, where

```
(deftemplate domain-effect
 (slot name (type SYMBOL)
  (default-dynamic (gensym*)))
 (slot part-of (type SYMBOL))
 (slot predicate (type SYMBOL))
 (multislot param-names)
 (multislot param-values)
 (multislot param-constants)
 (slot type (type SYMBOL)
  (allowed-values POSITIVE NEGATIVE))
)
```

Listing 5: The template definition for an action effect.

```
(:action enter-field
 :parameters (
  ?r - robot ?team-color - team-color)
 :precondition (and
  (location-free START INPUT)
  (robot-waiting ?r))
 :effect (and (entered-field ?r)
  (at ?r START INPUT)
  (not (location-free START INPUT))
  (not (robot-waiting ?r)) (can-hold ?r))
)
```

Listing 6: The PDDL operator `enter-field`.

the names must be a subset of the parameter names of the operator.

An action's *effect* is assumed to be a set of literals similar to STRIPS effects. The definition of the template is shown in Listing 5. Similar to preconditions, the effect name is automatically set to a unique name if no name is given. An effect must always be part of an operator and refer to a predicate. Similar to an atomic precondition, it has parameter names, values, and constants. An effect can be positive or negative.

The translation of the `enter-field` PDDL operator (Listing 6) is shown in Listing 7. The precondition of the PDDL action is represented by a conjunctive **`domain-precondition`** and two template facts of type **`domain-atomic-precondition`**. Similarly, the effect of the action is split into five instances of **`domain-effect`**, one for each atomic effect. The precondition `enter-field11` on the predicate `location-free` shows how constants are translated: The multislot `param-names` contains the two placeholder names `c`, indicating constant values. The multislot `param-constants` is set to `(START INPUT)`. Note that the parameter name is not used but is only a placeholder so the number of parameters of the precondition matches the number of parameters of the predicate. If a parameter is not a constant, then the respective value in `param-constants` is set to **`nil`**, as shown in the effect `gen65` on the predicate `at`.

5 Planner Integration

The handling of the planning system is implemented as a separate planner component. It handles PDDL problem generation, invokes a planner, and parses the output to re-

```
(domain-operator (name enter-field)
 (param-names r team-color)
 (param-types robot team-color))
(domain-precondition (part-of enter-field)
 (name enter-field1) (type conjunction))
(domain-atomic-precondition (part-of enter-field1)
 (name enter-field11) (predicate location-free)
 (param-names c c) (param-constants START INPUT))
(domain-atomic-precondition (part-of enter-field1)
 (name enter-field12) (predicate robot-waiting)
 (param-names r) (param-constants nil))
(domain-effect (name gen64) (part-of enter-field)
 (predicate entered-field) (param-names r))
(domain-effect (name gen65) (part-of enter-field)
 (predicate at) (param-names r c c)
 (param-constants nil START INPUT))
(domain-effect (name gen66) (part-of enter-field)
 (predicate location-free) (param-names c c)
 (param-constants START INPUT)
 (type NEGATIVE))
(domain-effect (name gen67) (part-of enter-field)
 (predicate robot-waiting) (param-names r)
 (type NEGATIVE))
(domain-effect (name gen68) (part-of enter-field)
 (predicate can-hold) (param-names r))
```

Listing 7: The operator `enter-field` in CLIPS (default values are omitted).

trieve the plan. It therefore provides an abstraction of the underlying PDDL planner. The component currently supports FF (Hoffmann and Nebel 2001) and FASTDOWNWARD (Helmert 2006), among others. As these planners produce sequential plans, we focus on sequential planning. However, the framework can be easily extended to partially ordered plans with a modified action selection (see Section 6.1).

Planner Invocation The executive continuously synchronizes the world model with a robot memory based on MongoDB (Niemueller, Lakemeyer, and Srinivasa 2012). The planner model (as part of the world model) is therefore available in the database. To initiate a planning process, the executive stores the goal to the robot memory and invokes the planner. The planner retrieves model and goal and dynamically generates the PDDL problem from using a (domain-specific) template. This way, the planner always plans with the same initial state as the CLIPS agent currently operates with. The PDDL domain is static and is the same domain that is parsed by the CLIPS agent. The CLIPS function `(pddl-call ?goal)` initiates this process.

The planner stores the generated plan in the robot memory, from which the executive retrieves it. It then asserts a **`plan`** fact along with a number of **`plan-action`** facts.

Action Grounding We differentiate operator definitions and instances thereof, i.e., grounded actions. A grounded action is defined by the template **`plan-action`**, as shown in Listing 8. A **`plan-action`** has a unique numeric `id`, which is used to impose an ordering of the actions in a plan.

```
(deftemplate plan-action
  (slot id (type INTEGER))
  (slot action-name (type SYMBOL))
  (multislot param-names)
  (multislot param-values)
  (slot status (type SYMBOL)
   (allowed-values FORMULATED PENDING
    WAITING RUNNING EXECUTION-SUCCEEDED
    SENSED-EFFECTS-WAIT
    SENSED-EFFECTS-HOLD EFFECTS-APPLIED
    FINAL EXECUTION-FAILED FAILED))
  (slot executable (type SYMBOL)
   (allowed-values FALSE TRUE))
)
```

Listing 8: The template definition for an action.

```
(defrule domain-check-atomic-precondition
  ?precond <-
    (domain-atomic-precondition
       (is-satisfied FALSE)
       (grounded TRUE)
       (predicate ?pred)
       (param-values $?params))
  (domain-fact (name ?pred)
               (param-values $?params))
=>
  (modify ?precond (is-satisfied TRUE))
)
```

Listing 9: The rule to check whether an atomic precondition is satisfied.

The slot `action-name` refers to a **`domain-operator`**, `param-names` and `param-values` denote the parameters of the action. The possible states are detailed in Section 6.

Given a **`plan-action`** fact, we need to ground the action's precondition to check whether the action is executable. In order to do so, an atomic precondition is grounded by matching the parameter names in the precondition with the parameter names of the grounded action and copying their values from the action to the precondition. A non-atomic precondition is grounded by grounding all its sub-conditions.

After grounding the action's precondition and all the sub-conditions recursively, we can check whether an action is executable. Starting with the atomic preconditions, we check whether a corresponding **`domain-fact`** with the same predicate name and the same parameter values exists. If so, the atomic precondition is satisfied. The CLIPS rule doing this check is shown in Listing 9. We then proceed with the parent preconditions: If the parent is a negation, then it is satisfied if and only if its child is not satisfied. If it is a conjunction, then all the children must be satisfied. We continue bottom-up until the root precondition is reached. If the root is satisfied, then the action is executable.

6 Plan Execution and Monitoring

Based on the grounded plan, the executive starts evaluation of the plan. Note that after each action execution, the remaining plan actions are grounded again, based on the then available information in the planner model. In the following, we describe the execution procedure in more detail.

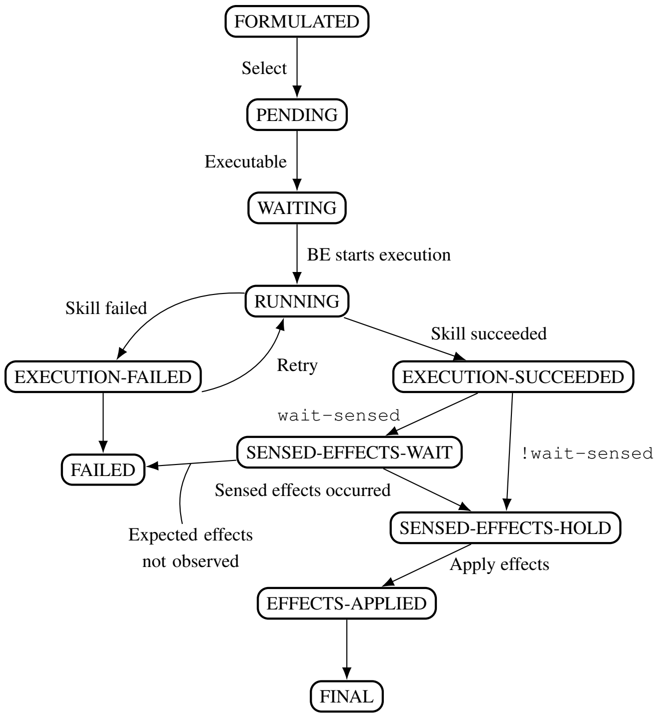

Figure 1: The possible states of a `plan-action` and the transitions between those states.

6.1 Plan Execution

Each action in the plan is assigned a state machine as shown in Figure 1. Initially, all actions of the plan are set in the `FORMULATED` state. The *action selection* is responsible for selecting the action to execute, changing its state from `FORMULATED` to `SELECTED`. The current version of the system supports sequential plans. When no action is being executed, the next action is determined by the action which is executable, and for which there is no other pending action with a lower `id` (cf. Section 5). If no pending action is executable, the agent waits until an exogenous event causes an update to the planner model rendering an action executable (also cf. Section 6.2).

Any action in the current plan is checked whether it is executable by checking whether its precondition is satisfied. If a pending action is executable, the action is marked as `WAITING`. There are two sub-systems which can process such actions. The first is directly inside the executive, and is used in particular for communication acts. The second interacts with the physical world through the Lua-based Behavior Engine (BE) (cf. Section 3). The CX is initially configured with a domain-specific set of mappings from operators to skill execution strings that the BE can interpret.

Once the BE starts executing a skill, the state of the action changes to `RUNNING`. The executive then awaits for the skill to finish. Depending on the outcome of the skill execution, the state of the action is set to either EXECUTION-FAILED or EXECUTION-SUCCEEDED. An action may be retried (cf. Section 6.2), otherwise it is FAILED. If the execution was successful, then the next state depends on whether the executed action has any *sensed effects*, i.e., effects involving sensed predicate (see below). Eventually, the action transitions to SENSED-EFFECTS-HOLD, asserts all other effects in one transaction, and transitions to EFFECTS-APPLIED. This enables additional steps such as retracting all precondition groundings and where execution monitoring may perform an analysis of the outcome. Then, the action transitions to FINAL.

Sensed Effects Some predicates may be configured to be *sensed predicates*, i.e., predicates which are set by exogenous action not under the control of the agent. This is a deliberate extension in the execution model, as it covers an important aspect often found in real-world domains, especially in robotics. During planning, such a predicate is treated as any other predicate. Actions, for which its effects involve sensed predicates, can be configured in two ways. They may either wait for those effects (sensed-wait slot is set to TRUE) or simply ignore the effects altogether. If the action waits for the effects in the SENSED-EFFECTS-WAIT state, the executive monitors updates to the planner model. Once all expected sensed effects are observed, the action transitions to the SENSED-EFFECTS-HOLD mode. If this never happens (i.e., an expected effect never occurs), execution monitoring will eventually let the action fail (see Section 6.2). If no sensed effect exists or if the action is configured not to wait for sensed effects, then the action state is directly set to SENSED-EFFECTS-HOLD. After that, the non-sensed effects of the action are applied by grounding the operator effects with the action's arguments, and then asserting and retracting **domain-fact** facts.

Note that if an action is configured to not wait on sensed effects, then the semantics of the action execution and the PDDL domain model may differ. In particular, an effect that is specified in the PDDL domain does not necessarily hold after executing the action if it is a sensed effect and the action does not wait on the sensed effect. However, this is useful for actions that have a delayed effect, e.g., an instruction message to a machine that eventually finishes processing the instruction and switches to a state READY. While this effect needs to be modeled in the domain to allow reasoning about the agent's actions, we do not want to wait for the machine to finish processing. Instead, the agent should continue with the plan until one of the action's requires the machine to be in the state READY, in which case the agent will wait until the precondition is satisfied.

An execution trace of a plan including action grounding, precondition check, and effect application, is shown in Figure 2 in the Appendix.

6.2 Execution Monitoring

Execution monitoring (XM) is the process of observing the system while performing an action. It is an important aspect of robust execution systems. The CLIPS Executive is particularly well-suited for this task since all information is available in the fact base common to all parts of the program. The explicit modeling of plans and action execution states provide the execution monitoring with integration hooks. For example, if an action enters the EXECUTION-FAILED state, based on additional information the XM may or may not indicate to retry an action. By default, it tries three times. It can also easily impose temporal monitoring, for example if the agent is stuck for a certain period in time without the next action to be executable, the XM can determine the plan to have failed and trigger re-planning.

7 Conclusion

In this paper, we introduce a PDDL plan executive based on the CLIPS rule-based production system. The executive is capable of executing a PDDL plan by invoking a planner, translating the plan into its internal plan representation, checking each action's executability, executing an action by means of a Behavior Engine, and applying effects based on the execution model. We provided a detailed description of the domain and plan representation used by the executive and described four different models for the integration of a PDDL planner. The domain model describes the operators, predicates, and object types of the PDDL domain. The execution model extends the domain model by additional aspects of action execution such as delayed effects, exogenous actions, and sensed predicates. The world model contains all relevant information known about the environment, while the planner model is a subset of the world model only concerned with the facts and objects necessary for planning. During execution, the domain model allows to verify that the current plan continues to be executable by checking the actions' preconditions, while continuously updating the world model based on sensing. If an action fails or an unexpected event occurs, the CX provides monitoring capabilities to recover, which is aided by the explicit modeling of plans, actions, and action execution states.

Acknowledgments

T. Niemueller was supported by the German National Science Foundation (DFG) research unit *FOR 1513* on Hybrid Reasoning for Intelligent Systems (http://www.hybrid-reasoning.org).

T. Hofmann was supported by the German National Science Foundation (DFG) grant *GL-747/23-1* on Constraint-based Transformations of Abstract Task Plans into Executable Actions for Autonomous Robots.

We thank the anonymous reviewers for their insightful comments and questions which helped clarify several aspects of this paper.

References

Cashmore, M.; Fox, M.; Long, D.; Magazzeni, D.; Ridder, B.; Carrera, A.; Palomeras, N.; Hurtos, N.; and Carreras, M. 2015. ROSPlan: Planning in the robot operating system. In *25th Int. Conf. on Automated Planning and Scheduling*.

Claßen, J.; Röger, G.; Lakemeyer, G.; and Nebel, B. 2012. PLATAS — integrating planning and the action language Golog. *KI - Künstliche Intelligenz* 26(1).

De Giacomo, G.; Lespérance, Y.; and Levesque, H. J. 2000. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence* 121.

De Giacomo, G.; Reiter, R.; and Soutchanski, M. 1998. Execution Monitoring of High-Level Robot Programs. *Proceedings of the 6th International Conference on Knowledge Representation and Reasoning (KR).*

Forgy, C. L. 1982. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence* 19(1).

Fox, M., and Long, D. 2006. Modelling Mixed Discrete-continuous Domains for Planning. *Journal for Artificial Intelligence Research* 27(1).

Giarratano, J. C. 2007. *CLIPS Reference Manuals.* `http://clipsrules.sf.net/OnlineDocs.html`.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research (JAIR)* 26.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)* 14.

Hofmann, T.; Niemueller, T.; Claßen, J.; and Lakemeyer, G. 2016. Continual Planning in Golog. In *Proceedings of the 30th Conference on Artificial Intelligence (AAAI).*

Ingrand, F.; Chatila, R.; Alami, R.; and Robert, F. 1996. PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots. In *IEEE Int. Conf. on Robotics and Automation (ICRA)*, volume 1.

Kim, P.; Williams, B. C.; and Abramson, M. 2001. Executing reactive, model-based programs through graph-based temporal planning. In *17th International Joint Conference on Artificial Intelligence (IJCAI).*

Levesque, H. J.; Reiter, R.; Lesperance, Y.; Lin, F.; and Scherl, R. B. 1997. GOLOG: a logic programming language for dynamic domains. *Journal of Logic Programming* 31(1-3).

McCarthy, J. 1963. Situations, actions, and causal laws. Technical report, DTIC Document.

Muñoz, P.; R-Moreno, M. D.; and Castaño, B. 2010. Integrating a pddl-based planner and a plexil-executor into the ptinto robot. In *Trends in Applied Intelligent Systems (IEA/AIE).*

Niemueller, T.; Ferrein, A.; Beck, D.; and Lakemeyer, G. 2010. Design Principles of the Component-Based Robot Software Framework Fawkes. In *Int. Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR).*

Niemueller, T.; Zwilling, F.; Lakemeyer, G.; Löbach, M.; Reuter, S.; Jeschke, S.; and Ferrein, A. 2016. *Industrial Internet of Things: Cybermanufacturing Systems.* Springer. chapter Cyber-Physical System Intelligence – Knowledge-Based Mobile Robot Autonomy in an Industrial Scenario.

Niemueller, T.; Lakemeyer, G.; Leofante, F.; and Abraham, E. 2017. Towards clips-based task execution and monitoring with smt-based decision optimization. In *Workshop on Planning and Robotics (PlanRob) at International Conference on Automated Planning and Scheduling (ICAPS).*

Niemueller, T.; Ferrein, A.; and Lakemeyer, G. 2009. A Lua-based Behavior Engine for Controlling the Humanoid Robot Nao. In *RoboCup Symposium 2009.*

Niemueller, T.; Lakemeyer, G.; and Ferrein, A. 2013. Incremental Task-level Reasoning in a Competitive Factory Automation Scenario. In *AAAI Spring Symposium - Designing Intelligent Robots: Reintegrating AI.*

Niemueller, T.; Lakemeyer, G.; and Srinivasa, S. 2012. A Generic Robot Database and its Application in Fault Analysis and Performance Evaluation. In *IEEE International Conference on Intelligent Robots and Systems (IROS).*

Reiter, R. 2001. *Knowledge in action: logical foundations for specifying and implementing dynamical systems.* MIT Press.

Roberts, M.; Alford, R.; Shivashankar, V.; Leece, M.; Gupta, S.; and Aha, D. W. 2016. ACTORSIM: A toolkit for studying goal reasoning, planning, and acting. In *WS on Planning and Robotics (PlanRob) at International Conference on Automated Planning and Scheduling (ICAPS).*

Schaepers, B.; Niemueller, T.; Lakemeyer, G.; Gebser, M.; and Schaub, T. 2018. Asp-based time-bounded planning for logistics robots. In *International Conference on Automated Planning and Scheduling (ICAPS).*

Verma, V.; Jónsson, A.; Pasareanu, C.; and Iatauro, M. 2006. Universal Executive and PLEXIL: Engine and Language for Robust Spacecraft Control and Operations. In *AIAA Space*.

Williams, B. C.; Ingham, M. D.; Chung, S. H.; and Elliott, P. H. 2003. Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers. *Proceedings of the IEEE* 91(1).

Wygant, R. M. 1989. CLIPS: A powerful development and delivery expert system tool. *Computers & Industrial Engineering* 17(1–4).

Appendix

```
 1 ==> f-1199  (domain-fact (name location-free) (param-values START INPUT))
 2 ==> f-1286  (domain-fact (name robot-waiting) (param-values R-1))
 3 ==> f-17314 (plan-action (id 2) (action-name enter-field) (param-names r team-color) (param-values R-1 CYAN))
 4 FIRE  221 domain-ground-action-precondition: *,f-17314,f-553,*
 5 ==> f-17315 (domain-precondition (part-of enter-field) (grounded-with 2) (name enter-field1) (type conjunction))
 6 FIRE  222 domain-ground-atomic-precondition: *,f-17314,f-17315,f-555,*
 7 ==> f-17316 (domain-atomic-precondition (part-of enter-field1) (grounded-with 2) (name enter-field12) (predicate
       robot-waiting) (param-names r) (param-values R-1))
 8 FIRE  223 domain-check-if-atomic-precondition-is-satisfied: f-17316,f-1286
 9 <== f-17316 (domain-atomic-precondition (name enter-field12) (is-satisfied FALSE))
10 ==> f-17317 (domain-atomic-precondition (name enter-field12) (is-satisfied TRUE))
11 FIRE  224 domain-ground-atomic-precondition: *,f-17314,f-17317,f-554,*
12 ==> f-17318 (domain-atomic-precondition (part-of enter-field1) (grounded-with 2) (name enter-field11) (predicate
       location-free) (param-names c c) (param-values START INPUT) (param-constants START INPUT) (is-satisfied FALSE))
13 FIRE  225 domain-check-if-atomic-precondition-is-satisfied: f-17318,f-1199
14 <== f-17318 (domain-atomic-precondition (name enter-field11) (is-satisfied FALSE))
15 ==> f-17319 (domain-atomic-precondition (name enter-field11) (is-satisfied TRUE))
16 FIRE  226 domain-check-if-conjunctive-precondition-is-satisfied: f-17315,*,*
17 <== f-17315 (domain-precondition  (name enter-field1) (type conjunction) (is-satisfied FALSE))
18 ==> f-17320 (domain-precondition  (name enter-field1) (type conjunction) (is-satisfied TRUE))
19 FIRE  227 domain-check-if-action-is-executable: f-17314,f-17320
20 <== f-17314 (plan-action (id 2) (action-name enter-field)  (status FORMULATED) (executable FALSE))
21 ==> f-17321 (plan-action (id 2) (action-name enter-field)  (status FORMULATED) (executable TRUE))
22 FIRE  357 action-selection-select: f-17321,f-17075,f-17102,*,*
23 <== f-17321 (plan-action (id 2) (action-name enter-field) (status FORMULATED) (executable TRUE))
24 ==> f-17678 (plan-action (id 2) (action-name enter-field) (status PENDING) (executable TRUE))
25 FIRE  358 skill-action-start: f-17678,f-295,*,f-1419
26 Calling skill 'drive_into_field{team="CYAN"}'
27 <== f-17678 (plan-action (id 2) (action-name enter-field) (status PENDING) (executable TRUE))
28 ==> f-17680 (plan-action (id 2) (action-name enter-field) (status WAITING) (executable TRUE))
29 ClipsExecutiveThread wants me to execute 'drive_into_field{team="CYAN"}'
30 GOTO: executing goto{place = C-ins-out}
31 Skill enter-field is S_RUNNING, was: S_IDLE
32 Action enter-field is running
33 <== f-17680 (plan-action (id 2) (action-name enter-field) (status WAITING) (executable TRUE))
34 ==> f-17685 (plan-action (id 2) (action-name enter-field) (status RUNNING) (executable TRUE))
35 Skill enter-field is S_FINAL, was: S_RUNNING
36 FIRE    3 skill-action-final: f-17681,f-17685,f-17859
37 Execution of enter-field completed successfully
38 <== f-17685 (plan-action (id 2) (action-name enter-field) (status RUNNING) (executable TRUE))
39 ==> f-17860 (plan-action (id 2) (action-name enter-field) (status EXECUTION-SUCCEEDED) (executable TRUE))
40 FIRE    4 domain-effects-check-for-sensed: f-17860,f-552
41 <== f-17860 (plan-action (id 2) (action-name enter-field) (status EXECUTION-SUCCEEDED) (executable TRUE))
42 ==> f-17861 (plan-action (id 2) (action-name enter-field) (status SENSED-EFFECTS-HOLD) (executable TRUE))
43 FIRE    5 domain-effects-apply: f-17861,f-552
44 ==> f-17862 (domain-fact (name entered-field) (param-values R-1))
45 ==> f-17863 (domain-fact (name at) (param-values R-1 START INPUT))
46 <== f-1199  (domain-fact (name location-free) (param-values START INPUT))
47 <== f-1286  (domain-fact (name robot-waiting) (param-values R-1))
48 ==> f-17864 (domain-fact (name can-hold) (param-values R-1))
49 <== f-17861 (plan-action (id 2) (action-name enter-field) (status SENSED-EFFECTS-HOLD) (executable TRUE))
50 ==> f-17865 (plan-action (id 2) (action-name enter-field) (status EFFECTS-APPLIED) (executable TRUE))
51 <== f-17865 (plan-action (id 2) (action-name enter-field) (status EFFECTS-APPLIED) (executable TRUE))
52 ==> f-17915 (plan-action (id 2) (action-name enter-field) (status FINAL) (executable TRUE))
```

Figure 2: An abbreviated execution trace for executing a plan that contains the action `enter-field`. The initial world model is shown in lines 1-2. In lines 4-21, the precondition is grounded and checked and the action is marked as `executable`. In lines 22-39, the action is executed with the Behavior Engine. In lines 40-51, the effects of the action are applied. At the end, the action is marked as `FINAL`.