Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)

Goal Reasoning in the CLIPS Executive for Integrated Planning and Execution

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

Abstract

The close integration of planning and execution is a challenging problem. Key questions are how to organize and explicitly represent the program flow to enable reasoning about it, how to dynamically create goals from run-time information and decide on-line which to pursue, and how to unify representations used during planning and execution.

In this work, we present an integrated system that uses a goal reasoning model which represents this flow and supports dynamic goal generation. With an explicit world model representation, it enables reasoning about the current state of the world, the progress of the execution flow, and what goals should be pursued – or postponed or abandoned. Our executive implements a specific goal lifecycle with compound goal types that combine sub-goals by conjunctions, disjunctions, concurrency, or that impose temporal constraints.

Goals also provide a frame of reference for execution monitoring. The current system can utilize PDDL as the underlying modeling language with extensions to aid execution, and it contains well-defined extension points for domain-specific code. It has been used successfully in several scenarios.

1 Introduction

Robotics remains to be a challenging environment for task planning and scheduling. Typical domains often require elaborate and intricate modeling, hybrid numeric symbolic representations, temporal constraints, and a notion of uncertainty. Large efforts are made to simplify the actual problem enough so it can be expressed in a frugal language such as PDDL. Planning times are a particular challenge, as often a near-reactive performance is expected and long planning times are hardly acceptable. Robots are machines that are intended to achieve a specific objective in the physical world. This emphasizes the particular importance of the actual execution of plans. Problem sizes can scale from static robot arm, over mobile transport platforms, to complex multi-robot scenarios, that require a great many decisions along the implementation and integration process, e.g., whether to use a central planner, or a distributed approach with local decision making and coordination.

Many languages and systems to represent and facilitate the execution of such plans have been proposed in the past, for example OpenPRS, PLEXIL, RMPL, or ROSPlan, to name but a few (cf. Related Work in Section 6). PLEXIL and RMPL focus mostly on a plan representation language with associated executives which can interpret such plans and invoke actions. OpenPRS provides a reasoning engine and plan representation language, but lacks a powerful planner integration. ROSPlan provides a fully integrated system that incorporates a knowledge base, an accessible domain representation, planner integration, and an execution system. However, for such systems goals are mostly static and given as part of the invocation of that very system, rather than being an area of reasoning itself.

In this paper, we present the CLIPS Executive (CX), an integrated system that performs the entire high-level decision making process. The CX uses an explicit representation of goals and their dependencies as its core structure to guide and monitor the overall program flow. This is based on the idea of goal reasoning (Aha 2018), which means to continually reason about goals to pursue and dynamically adjust or change goals. In the CX, goals adhere to a specific lifecycle extending the ideas by Roberts et al. (2014), which makes the high-level execution flow explicit and observable, and can provide explanations of what the robot is doing and why. A declarative, domain-specific component reasons about and formulates relevant goals, and makes decisions on which goals to select and pursue, or when to abandon a goal. Goals can be organized in tree structures with expressive inner node semantics. This way, planning problems can be decomposed into more manageable sub-problems. This is done, for one, to improve planning performance, and for another to enable the encoding and exploitation of known structure inherent to the domain. To bring about a goal, off-the-shelf planners can be used to generate a suitable plan, for example PDDL-based planners. We avoid modeling inconsistencies, that can often arise between a planner and the executive, by using the exact same PDDL domain model for both purposes, and augmented it with extensions relevant for execution. The CX can execute plans according to a user-specific action selection strategy, e.g., sequentially or concurrently with temporal constraints. Representing goals explicitly also enables generic and domain-specific execution monitoring. This can be on a very coarse level, e.g., purging infeasible goals, or very fine-grained probing down into the action representation of plans and world model facts.

Copyright © 2019, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Our main contributions are the fine-grained modeling of all aspects of the planning and execution system, the close integration between planning and execution models, and the explicit flow definition through a revised goal lifecycle. To the best of our knowledge, there is no such comprehensive (goal) reasoning and execution system thus far.

An example domain is the Planning and Execution Competition for Logistics Robots in Simulation (PExC) (Niemueller et al. 2016a). There, a group of three robots has to plan, execute, and coordinate production in a virtual factory. Challenges are the temporal constraints on orders, which must be delivered in specific time windows, concurrency requirements, uncertainty, the combinatorial challenge during planning, and oversubscription (more orders come in than are feasible, and the robot team must make decisions which orders to pursue).

In the following, we give a brief overview of our system (Section 2), before giving details about the goal representation and how it is used for flow control (Section 3). In Section 4 we describe planning and execution. We give a brief evaluation in Section 5. Detailed related work can be found in Section 6. We conclude in Section 7.

2 System Overview

The CX is implemented using the CLIPS rule-based production system (Wygant 1989). It consists of a fact base, a set of rules, and a forward chaining inference engine. Rules describe patterns in the fact base, which trigger its activation when matching using the efficient graph-based Rete algorithm (Forgy 1982). When a rule is activated, it is added to an agenda. A conflict resolution mechanism singles out one activated rule on the agenda and executes its body. This is done until no more rules are on the agenda.

CLIPS was chosen because it provides a very efficient pattern matching engine which fits nicely with the overall event-based flow. This is also what enables to easily and efficiently scale to a large number of goals, deep trees, and large plans, since the individual rules need only be concerned with an element at the appropriate level, e.g., a goal or a plan action, without getting lost in the explicit specification of the actual control loop or even parallelization. Furthermore, it generates extensive logs describing exactly when and especially why a certain rule fired, and what modifications it made to the knowledge base. This detailed tracing enables in-depth temporal and causal evaluation and debugging (with some tool support).[1]

The CX emphasizes a *separation of concerns*. The following components can be distinguished:

Domain and plan representation The domain is represented explicitly in the fact base. It is based on PDDL input, but features some extensions relevant during execution (Section 2.1). The plan representation supports durative actions.

Goal reasoning A domain-specific set of rules determines goal mode transitions (cf. Section 3.4).

[1] This has been used, for example, for a detailed automated analysis (Niemueller et al. 2015) of logistic robots instructed by a CLIPS-based controller and monitor (Niemueller et al. 2016b).

Action selection This component enables diverse interpretations of plans, for example executing a plan in sequence, or considering temporal dependencies enabling concurrency.

Action execution Executor modules implement or invoke an action, e.g., calling a basic behavior, a ROS action, or starting a communication act.

Coordination Handling limited resources and ensuring a conflict-free execution for a group of agents may require a module to communicate and provide mechanisms such as leader election or mutual exclusion.

World model and state estimation The current belief about the world must be kept consistent and synchronized with other agents. It also provides the one binding source of knowledge. It also requires incorporating information from the underlying system or received over the network.

Execution monitoring Representing goals, plans, and their progress explicitly enables reasoning about the current state of affairs, in particular validating the remaining plan with respect to the current world model (cf. Section 4.4).

These components are implemented in CLIPS in separate areas. The interface between the components is defined by a specific set of facts in the common shared fact base. For example, the goal reasoner reacts on goal fact changes (cf. Section 3.1) and draws on the common world model representation in the fact base. The action selector handles plan actions defined in plans associated with a dispatched goal, and issues commands to the underlying system. Multiple systems communicate through a replicated database (robot memory). The CLIPS context is embedded into the system's main loop typically running at 25 Hz.

2.1 Models

In the following, a number of different models with varying scopes are necessary for the description of the PDDL integration. We briefly describe each of these models (Niemueller, Hofmann, and Lakemeyer 2018).

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. Section 4.3).

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. 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.

Models P and W are synchronized automatically. The system tracks changes in either model and replicates the necessary modifications in the other model. This synchronization is performed with high priority, avoiding making decisions on a partly updated model.

PDDL has been chosen as a baseline representation for its simplicity and widespread use. As long as appropriate synchronization mechanisms are provided, it can be exchanged or extended. Another option is to provide appropriate conversion methods to integrate planners with other models. We have, for example, integrated the CX with an SMT-based planner (Niemueller et al. 2017) where we generate an appropriate SMT representation for planning from the CX world model.

3 Goals and a Goal Lifecycle as Flow Control

Goals are the core entity that describes objectives which are considered or pursued. In the following, we give a detailed account of goals (Section 3.1), how they are refined over time (Section 3.2), how to compose them in tree structures (Section 3.3), and how to change goal modes (Section 3.4).

3.1 Goals

Goals are one of the core data structures and describe all relevant aspects to achieve certain objectives in a declarative way. A goal is either meant to *achieve* or to *maintain* some condition or state. Each goal instance has a unique identifier and a priority. A crucial property to describe the program flow is the *goal mode*, which describes what has happened to the goal last. Once a goal has advanced sufficiently, it carries information about its outcome. Goals can have parameters and meta information that allow for easy re-use of goal templates. Furthermore, goals can be used to initiate and track resource allocation.

Listing 1 shows the data structure used to represent a goal. Each goal has a unique identifier and its type. The class is used to match goal reasoner rules (cf. Section 3.4). The sub-type and parent slots are used to form goal trees. The mode and outcome slots are used for flow control. Error information can be provided in machine and human readable form. Parameters enable to use goal templates for goal instantiation, and the meta field is used as an internal storage, e.g., to count the number of retries. There can also be resource requirements and allocations associated with a goal (cf. Section 4.5). Once the executive has committed to a goal, it denotes which plan or sub-goal should be executed.

```
1  (deftemplate goal
2    (slot id (type SYMBOL))
3    (slot class (type SYMBOL))
4    (slot type (type SYMBOL) (default ACHIEVE)
5               (allowed-values ACHIEVE MAINTAIN))
6    (slot sub-type (type SYMBOL))
7    (slot parent (type SYMBOL))
8    (slot committed-to (type SYMBOL))
9
10   (slot mode (type SYMBOL)
11              (allowed-values FORMULATED SELECTED
12                              EXPANDED COMMITTED
13                              DISPATCHED FINISHED
14                              EVALUATED RETRACTED))
15   (slot outcome (type SYMBOL)
16                 (allowed-values UNKNOWN COMPLETED
17                                 FAILED REJECTED))
18   (multislot error)
19   (slot message (type STRING))
20
21   (multislot params)
22   (multislot meta)
23   (multislot required-resources (type SYMBOL))
24   (multislot acquired-resources (type SYMBOL))
25 )
```

Listing 1: Goal template in CLIPS.

3.2 Goal Lifecycle

The goal lifecycle describes how a goal progresses over time, which is represented its mode. The lifecycle proposed in this work is based on ideas by Roberts et al. (2014), which is implemented in ActorSim (Roberts et al. 2016). In their lifecycle, there is a strict refinement order that determines how goals proceed. Once a goal has finished, it is evaluated. It may be pruned or moved back to an earlier mode.

We adapt the lifecycle by re-ordering certain steps, and accounting for explicit goal rejection, and a representation of the outcome of a goal. Figure 1 shows the proposed lifecycle. An *achievement* goal is initially *formulated*, only stating that it may be relevant and should be considered. By some reasoning mechanism, a goal may be *selected*. This will trigger *expansion*, for example invoking a planner, possibly to multiple goals at the same time. The CX may then *commit* to a plan or sub-goal of selected (non-conflicting) goals. The plans associated with such goals are then *dispatched*. Eventually, the goal is *finished*, after which the outcome is recorded to indicate that the goal has succeeded or failed to bring about the intended effects. The goal and its outcome is then *evaluated*, that is, the implications on the world model are determined and applied. Then, the goal is marked for *retraction*, which allows the CX to remove the goal, release resources (cf., Section 4.5), and any other associated data such as plans. Note the lifecycle's alternating acting and choice modes (indicated by gray and white boxes in Figure 1, respectively). Before committing to a goal, it may be *rejected*. A typical pattern is to generate a number of possible candidate goals. One or more goals are then selected for execution, the remaining goals are rejected. This clearly represents the intentions of the CX. Unlike the ear-

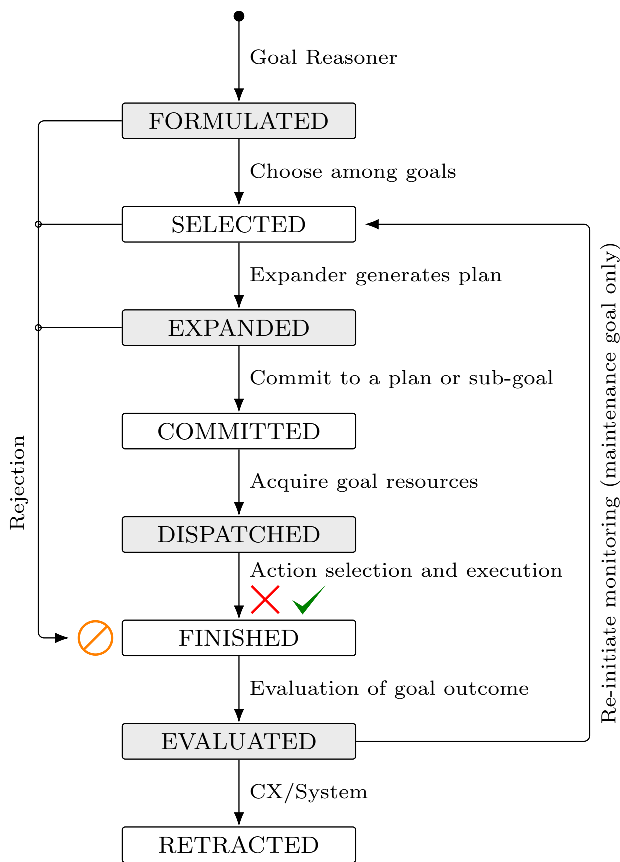

Figure 1: Goal lifecycle: each box represents a possible mode. Gray filled boxes denote modes for which the CX must perform some function, white ones are the result of making some choice.

lier model (Roberts et al. 2014), the CX goal lifecycle does not allow for an achievement goal that has progressed into a later goal mode to be reset into an earlier mode.

Figure 2 shows three example goal lifecycles. Goal 1 is an achievement goal. It is expanded with two plans (ensemble planning). The goal reasoner (cf. Section 3.4) commits to one of the plans for execution. It is then dispatched and eventually fails, due to the failure of some action. Goal 2 is expanded with a single plan. However, it is subsequently rejected, for example because some other goal was prioritized, or its requirements could not be met. Goal 3 is the root of a goal tree (cf. Section 3.3 for details). It is supposed to try all of its sub-goals and succeed if any of the sub-goal succeeds. Here, a new goal 3a is posted and eventually succeeds.

A *maintenance* goal is meant to act in several capacities: it denotes a constraint to be considered, for example, during goal selection; it may also describe a condition, which must hold. We focus in particular on the latter case. Such a condition may be used, for example, to assert the availability and health of some software component, or to implement periodic actions, such as service discovery announcements (where the condition would be invalidated periodically with a timeout). In this representation, the conditions can become a desired state. Once a maintenance goal is selected, it monitors the condition. If the condition is violated, the program is not halted, but rather the goal is expanded by creating a new achievement sub-goal, which is supposed to recover the very condition. Once the sub-goal has been evaluated, the maintenance goal finishes and may then be selected to enable monitoring again.

3.3 Goal Trees

Goals describe the objectives to achieve or conditions to maintain. Some such goals may be related in specific ways, creating dependencies among them. To model these, we introduce the concept of a *goal tree*. A goal tree recursively consists of a root goal, which denotes the specific handling of the tree, and one or more sub-goals. Goal 3 in Figure 2 is a tree root goal. During expansion, sub-goals are created, of which 3a is shown in more detail. It is a regular achievement goal that generates and executes a plan. Note, that for the root goal some transitions are performed automatically by the CX, e.g., committing to the highest priority sub-goal after expansion (the priorities can be dynamically assigned to enable reasoning about the order of execution at run-time). If no sub-goal had been added at all for an expanded root goal, it would fail. If all sub-goals are rejected, the root goal is rejected. The user handles the sub-goal lifecycle as required (it may be the root of another sub-tree). To dispatch the root goal, the highest priority sub-goal is selected. Once this has been finished and evaluated, the root goal is finished.

There are currently five root goal types specified. A *run-all* goal runs all sub-goals. It succeeds if all sub-goals succeed and fails if any sub-goal fails (thus forming a conjunction on the results). A *try-all* goal runs sub-goals until at least one sub-goal succeeded, or fails if all sub-goals have failed (forming a disjunction on the results). The *run-one* anticipates rejection of goals and runs the first non-rejected goal. The outcome of this sub-goal then directly determines the root goal result (this forms a case-based choice node). The latter two root goal types define higher level control constructs. The *retry* goal can be used to re-try a sub-goal a specified number of times if it fails. If the sub-goal succeeds within the given maximum number of tries, the root goal succeeds. The *timeout* goal executes the sub-goal with a specified time bound. If the sub-goal does not complete within that time bound, it is deemed to have failed. Otherwise, the outcome of the sub-goal is passed along.

Goal Trees and Similarities to 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 plan-

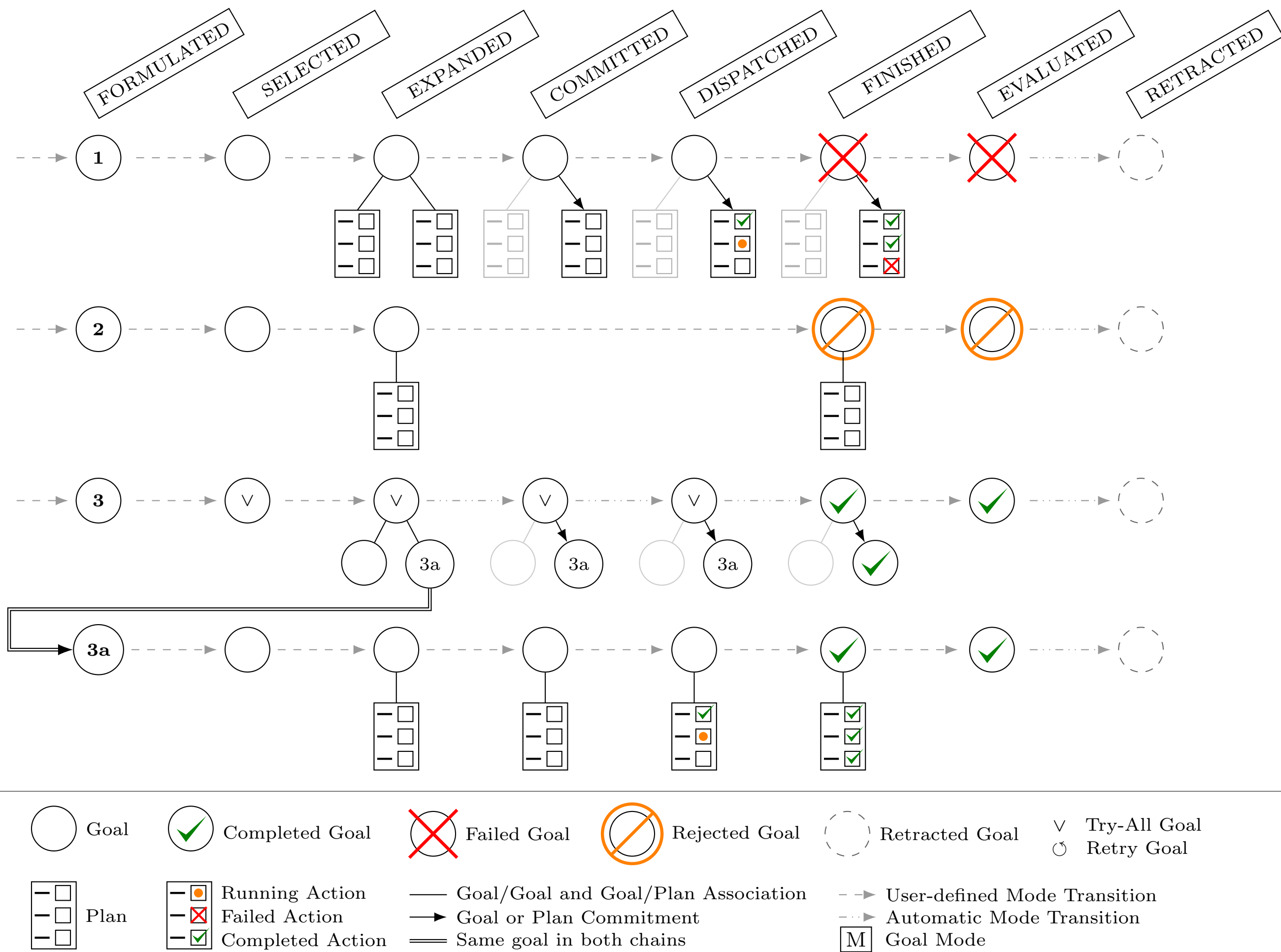

Figure 2: Several goals in-flight in the CX. The goal lifecycles may be independent and even run in parallel.

ner with PLEXIL (Muñoz, R-Moreno, and Castaño 2010).

There is a certain similarity between goal trees and PLEXIL nodes. While in PLEXIL, nodes provide semantics for an imperative instruction language, goals provide a declarative way to express objectives and intentions. Goal trees can lift plan execution instructions to defining mission policies.[2] However, at this point, we do not provide a full mapping from PLEXIL expressions or nodes to goal trees.

3.4 Goal Reasoning

Goals transition through several modes during their lifecycle. The CX component, which is responsible for triggering these mode transitions and making choices in the respective modes, is the *goal reasoner*. This is in accordance with Aha's definition: "goal reasoning [is] the process by which intelligent agents continually reason about the goals they are pursuing, which may lead to goal change" (Aha 2018).

[2]This is somewhat similar to the way Golog (Levesque et al. 1997) lifted imperative programming concepts from machine instructions to actions operating in an environment.

The CX provides a framework which allows a developer to express goals, how they change, and to formulate goal trees easily. The actual formulation of the CLIPS rules which define the very criteria for mode changes is highly domain-dependent, i.e., it is often based on domain-specific knowledge in the world model.

Listing 2 shows example rules that initiate planning once a PDDL goal has been selected, and associate a plan with a goal and mark it as expanded once the planner completes.

4 Planning and Execution

One particular way to expand a goal is to use a planner. Once the goal is dispatched, its plan needs to be executed and sensing results need to be incorporated while monitoring the execution of the plan. Additionally, the agent may need to coordinate with other agents to pursue shared goals and to handle limited resources.

4.1 Planner Integration

One method to expand a goal is to use a planner, such as a PDDL planner as shown in Figure 3. In order to plan a

```
1  (defrule goal-reasoner-pddl-planning-start
2    ?g <- (goal (id ?goal-id)
3                (class PDDL) (mode SELECTED))
4  =>
5    (pddl-start ?goal-id)
6  )
7
8  (defrule goal-reasoner-pddl-planning-finished
9    ?g <- (goal (id ?goal-id)
10               (class PDDL) (mode SELECTED))
11   ?c <- (pddl-done (goal ?goal-id) (plan ?plan-id))
12   ?p <- (plan (id ?plan-id))
13 =>
14   (retract ?c)
15   (modify ?p (goal-id ?goal-id))
16   (modify ?g (mode EXPANDED))
17 )
```

Listing 2: Goal expansion through planning.

task with a PDDL planner, we need a *domain* description and a *problem* description. As we also use PDDL to represent the actions in the domain model, re-using the PDDL domain description as planner model guarantees that the resulting plan will actually be executable and accomplish the goal. The problem description is generated from the planner model, which is a sub-set of the world model, as described in Section 2.1. The PDDL goal formula is specified in the expansion rule of the particular goal. After generating the problem description, a PDDL planner is called, which runs concurrently to the CX. This allows to plan for a goal while another goal is executing, or to plan for the same goal with multiple planners. Limiting the planner model to a sub-set of the relevant facts of the world model reduces the size of the search space of the planner and therefore increases planning efficiency, while guaranteeing consistency.

The goal is considered as expanded once the planner generated a plan. If the PDDL planner fails to determine a plan, the goal has *failed*. Following the goal lifecycle (cf. Section 3.2), the CX may or may not decide to commit to the goal, for example evaluating the plan's cost.

4.2 Plan Execution

In this paper, we focus on the execution for sequential plans (partially ordered plans with concurrent execution of multiple actions are, however, generally possible). Once committed, execution starts and the goal is marked as dispatched.

Each action of the plan follows the state machine shown in

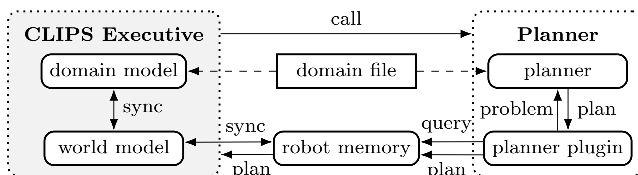

Figure 3: Data flow for PDDL-based planner integration and robot memory world model synchronization.

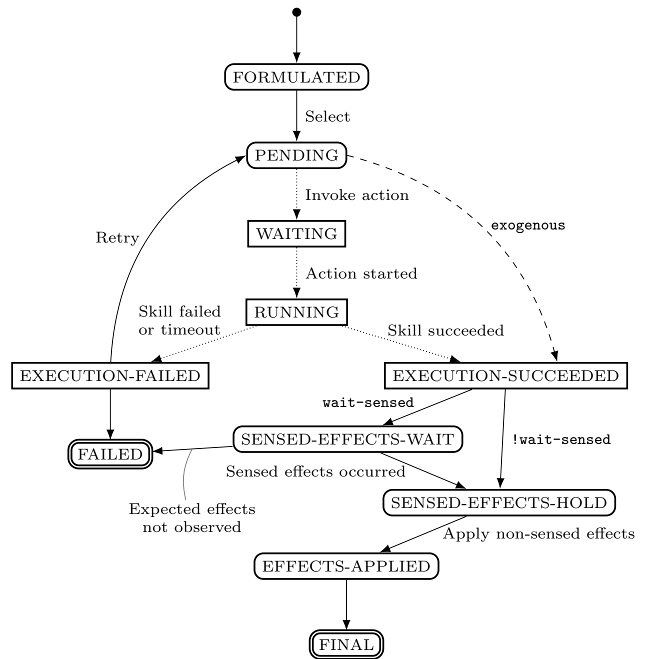

Figure 4: The possible states of a plan action and the transitions between those states.

Figure 4. In the first step, the *action selection* determines the next *executable* action to execute, which may be as simple as picking the next action in the plan if there is no action being executed. Note that while the action may be executable in the planner model, it may not be executable, yet, during action selection, as the occurrence of exogenous events may have caused its precondition to not be satisfied, yet. The selected action is marked as *pending* and passed to an executor, which controls the dotted transitions to the sharp cornered states. The CX currently supports two kinds of executors: the first is implemented inside the executive in CLIPS, e.g., for communication. The second executes a physical action through, e.g., the Lua-based Behavior Engine (Niemueller, Ferrein, and Lakemeyer 2009). Once the executor started the action, it is *waiting* for the base system to confirm the action is *running*, and eventually finishes. For a failed action, the goal its plan belongs to is typically considered failed. If successful, the effects are applied (for sensed effects cf. Section 4.3) and the action is set to *final* and the goal proceeds.

4.3 Sensing

Updated sensing results are directly incorporated into the world model, i.e., whenever a sensor reports new data, the respective world model fact is updated. If the sensing result is relevant for the planner model, the fact is also propagated. Whenever the planner model changes, the preconditions of all currently pending and waiting actions are re-evaluated. Thus, we can directly detect any change that can affect the plan that is currently being executed. This is particularly useful if we want to make use of *sensed effects*. A sensed effect of an action is *expected* to occur, but it is not immediately applied to the planner model. Instead, the CX waits

```
1  (defrule xm-next-action-not-executable
2    ?g <- (goal (id ?goal) (mode DISPATCHED)
3                 (committed-to ?plan))
4    (plan (id ?plan) (goal-id ?goal) (type SEQUENTIAL))
5    ; At least one action which is still to be executed
6    (plan-action (goal-id ?goal) (plan-id ?plan)
7                 (id ?id) (action-name ?action-name)
8                 (state FORMULATED) (executable FALSE))
9    ; No other FORMULATED action which comes earlier
10   (not (plan-action (goal-id ?goal) (plan-id ?plan)
11                     (state FORMULATED)
12                     (id ?oid&:(< ?oid ?id))))
13   ; there is no action which is currently ongoing
14   (not (plan-action (goal-id ?goal) (plan-id ?plan)
15                (state ~FORMULATED&~FINAL&~FAILED)))
16  =>
17   (modify ?g (mode FINISHED) (outcome FAILED)
18              (error STALLED-NONE-EXECUTABLE))
19 )
```

Listing 3: Execution monitoring for stalled sequential plans that should strictly progress.

for the world model update that confirms that the effect has occurred, and only then applies the remaining effects and sets the action to *final*. Optionally, instead of waiting for the effect to occur, we can also mark an action to not wait for the sensed effects, and instead directly continue with the execution of next step in the plan. This is useful especially for modeling actions that start an external process. For example, after pressing the button of a machine, instead of waiting for the machine to finish, we state that the effect will eventually occur and continue the execution of the remaining plan. Furthermore, waiting for a precondition on the same predicate in later actions ensures that the effect does happen in due time (or we can determine that it did not and abort).

Sensed effects are only part of the execution model and are not represented as such in the planner model. Instead, the planner treats them as normal effects.

4.4 Execution Monitoring

The execution of a plan needs to be monitored continuously to deal with exogenous events. We provide generic execution monitoring rules, e.g., to retry a failed action, or to fail an action if it takes too long. Additionally, one may want to let a goal fail early if the goal will no longer be accomplished by the current plan, or if the goal is not desirable anymore. In the first case, a re-expansion of the goal will result in an updated plan. Alternatively, instead of failing the goal, execution monitoring also allows to adapt the current plan in light of exogenous events, e.g., by adding actions that revert an exogenous effect. By doing so, the number of failed goals may be minimized, leading to a more efficient execution while not pursuing goals that are no longer useful.

Listing 3 shows an example for sequential plans to detect if a plan cannot progress when the preconditions of the next action are not met. Similarly, temporal plans can be monitored, e.g, for actions running over time or starting too late.

Handling Uncertainty Uncertainty is handled at execution time and not explicitly represented for planning, e.g., as probability models. Instead, it is handled as part of the execution monitoring. A close supervision is applied to both, goals and plans as they unfold and deviations can be detected quickly. An action with uncertain outcome could be repeated, or the goal re-expanded into a new plan. The division of goals into smaller sub-goals reduces the required re-planning time and thus the overall impact on execution, making this more feasible. A domain designer might also use a *try-all* goal (cf. Section 3.3), which tries several options, e.g., to account for possible failure. In the future, we may add an option for a more explicit representation of uncertainty, which could also be used during planning.

4.5 Multi-Agent Task Coordination

In *cooperative distributed planning*, a group of agents cooperates to fulfill a shared set of goals, whereas in *negotiated distributed planning*, each agent pursues its own goal and needs to coordinate with the other agents in order to render its local plan successful (DesJardins et al. 1999). We support both scenarios by implementing multi-agent world model synchronization, a generic locking procedure, and a resource locking mechanism that is tightly bound to goals.

Shared World Model Certain world model facts are not only useful locally, but also to other agents, e.g., the state of a commonly used machine. We implement world model synchronization using a shared database (Niemueller, Lakemeyer, and Srinivasa 2012). Each robot runs a database instance for local (agent-specific) and global (shared) world model facts. The global world model database is part of a replica set with one database instance for each robot. The replica set mechanisms take care of data distribution, shared updates, and conflict resolution. This way, if one agent updates a fact in the global world model, it is automatically distributed to the other agents, and the eventual consistency guarantees of the database propagate to the world model.

Mutual Exclusion For locking, we again built upon the replicated database, where we maintain a distinguished collection for mutex locks. An agent may lock a mutex by updating the lock owner in the respective database entry and requesting a *majority acknowledgment*, i.e., a majority of the agents have to agree to the update. This way, we can guarantee that at any point in time, at most one agent owns a specific lock. Based on this locking mechanism, we specify the two actions *lock* and *unlock*, which can be used as regular actions as part of a plan.

Consider the simple example of a *goto* action, as shown in Listing 4. By requiring a lock on the location before moving there, the planner will automatically add a lock action to the plan. This action will always succeed in the planner model. However, during execution, another agent may have already acquired the lock, breaking the precondition of the *lock* action. The action will fail to execute and the respective goal will be aborted, if desired. Alternatively, an implementation could wait for the lock for a certain amount of time.

```
1 (:action lock
2   :parameters (?m - mutex)
3   :precondition (not (locked ?m))
4   :effect (and (locked ?m)))
5 (:action goto
6   :parameters (?from ?to - location)
7   :precondition (and (at ?from) (locked ?to))
8   :effect (and (not (at ?from)) (at ?to)))
```

Listing 4: The actions *lock* and *goto* in PDDL

Resource Allocation Using mutex locks, we provide a goal-specific resource allocation mechanism. Each goal can have a number of *required resources* that it needs before it can be dispatched. If a goal has a required resource, the goal reasoner requests a lock for the resource once the goal is committed. Only after all required resources have been allocated, the goal changes its mode to *dispatched*. If a resource lock cannot be acquired because the resource is currently used by another agent, all already acquired resources are released and the goal is *rejected*. Once a goal is *retracted*, all its acquired resources are released.

The resource allocation mechanism allows a straightforward implementation of multi-agent coordination, since the domain designer only needs to specify the required resources. During execution, only one agent can successfully acquire all resources for a specific goal and is able to dispatch it, all other agents reject the goal and select another if applicable. A typical example would be a machine that a robot wants to use. Only one robot will acquire the machine resource, the other robots will select goals that do not involve that machine. By making the locking resource-specific rather than goal-specific, the agents will not only avoid pursuing the same goal, but will also reject any goal that involves a resource that is used by another agent.

In contrast to lock actions described above, a resource is always locked for the entire remaining lifecycle of the goal.

5 Evaluation

We have implemented several scenarios, ranging from low (single robot, few actions) to high complexity (multi-robot scenario, long running). Here, we focus on productions logistics competitions for real robots and in simulation.

The CX has been used since 2018 in the RoboCup Logistics League (Niemueller et al. 2013) by the Carologistics team. There, it is used with pre-defined plans using a generate-and-filter approach, where the robot formulates potential goals when idle, rejects infeasible ones, and then picks the highest priority/reward goal. With this system, the team was able to win the RoboCup German Open 2018.

For the PExC competition in simulation, the system has been combined with an SMT-based planning system (Niemueller et al. 2017). There, rather coarse goals (per order) are formulated and a heuristic selects a feasible one. Then, a suitable representation generated based on the CX' world model and passed to the planner. The resulting plan with explicit action dependencies is converted into a CX plan and executed. The approach won the PExC 2018 competition. The results are shown in Figure 5, with another approach based on Answer Set Programming. While the ASP-based planner provides a better density and higher scores, it is still based on an older system (cf. CLIPS-based Agent in Section 6) that provides less robustness. The CX with SMT has a better robustness and therefore much higher median scores.

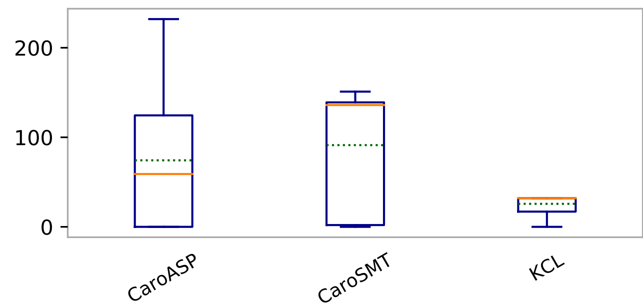

Figure 5: Scores of the PExC 2018 round-robin qualifiers (300 games). Box indicates the 25% and 75% quartiles, yellow line is median, dotted green line average, whiskers represent minimum and maximum scores. CaroASP and CaroSMT are ASP- and SMT-based planning systems of the Carologistics team, KCL is King's College London with ROSPlan.

The computational overhead of the CX itself is negligible compared to other software components of the robot, in particular planning systems.

6 Related Work

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

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 slightly different life cycle. ActorSim itself does not provide a ready-to-use integration for planning systems. Expansion generates a task, which is then executed through invoking external actions.

ActorSim as an implementation still has some rough edges, such as requiring to write reasoning processes in Java that must be compiled on every modification, or deep and indirect call chains making it harder to trace. We did implement the logistics scenario in ActorSim and ran hundreds of simulated games, in part with very good outcomes. Based on our experience with ActorSim we revised the goal lifecycle to more precisely capture the interactions observed in actual robot systems, and to rephrase what it means to evaluate a goal. We also added the explicit notion of rejection

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

and outcome, which is particularly important to enable, e.g., a generate-and-filter approach, which generates a number of candidate goals, and then sifts through them selecting the most relevant one(s) only avoiding conflicts.

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 plan dispatcher generates an Esterel program (Berry and Gonthier 1992) for execution. It represents actions as nodes and connects them through signals and slots. Actions are invoked on the base system for the active nodes. The dispatcher does not evaluate preconditions of actions during execution and hence may invoke actions which cannot be accomplished. Effects are not automatically applied to the fact base, but the external action provider has to.

CLIPS-based Agent The rule-based production system CLIPS has been used before in 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. The system has been extended with a multi-robot time-bounded planning based on Answer Set Programming (Schäpers et al. 2018).

The CX is an indirect successor of this system, redesigned from the ground up, putting more emphasis on providing exact modeling on each part of the system and explicitly representing the program flow, goal evolution, and interaction between planning and execution models.

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).[4] 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. Tasks are specified in terms of small programs (supporting loops, conditionals, and recursion), called OPs. OPs have logic formulas as activation conditions. A specialty is that multiple OPs can be executed in parallel, which can make proper plan design non-trivial due to race conditions. OpenPRS does not directly support planner integration, but typically OPs (partial plans) are written manually.

We have modeled a logistics scenario in OpenPRS (Niemueller et al. 2016c) and extended it to participate in PExC 2017. OpenPRS' parallel invocation pattern does pose some challenges to cleanly monitor activated OPs and its meta reasoning capabilities are somewhat tricky to use, which makes it harder to filter possible OPs. An additional problem for a generate-and-filter approach is the inability to phrase queries such as "find a goal, such that there is no goal with lower priority". The syntax can represent this, but the interpreter cannot evaluate such queries.

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

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 interleaved 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 (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 continual planning (Hofmann et al. 2016), which interleaves PDDL-based planning with GOLOG plan execution, plans for acquiring missing knowledge, and monitors the environment for unexpected events and exogenous actions.

T-REX T-REX (McGann et al. 2007) describes a communication protocol among goals. Several components can propose or process goals. Goals are expanded either into a timeline or a direct action. A specific reactor has exclusive ownership of a timeline. Other reactors can then operate on subgoals in a timeline. Global discrete time is used to synchronize the timelines. The information flow is modeled along the hierarchical structure of timelines.

The CX provides a more abstract representation of goals than T-REX, which can be reasoned about and used for execution monitoring. The CX' event-based flow enables a more flexible design and is clearer through its explicit goal modes. The multiple components in T-REX could be modeled as goal types in the CX, where expanders would operate on the individual goals. The CX is more strict in distinguishing goals and plans, where only plans contain direct actions.

7 Conclusion

In this paper, we have described the CLIPS-based Executive (CX) focusing on its goal reasoning capabilities. Its overall program flow is represented and organized by explicitly representing goals and their specific lifecycle. It determines how goals progress through several modes, where goal trees enable to impose more structure on the goal-level. The system is closely integrated with, e.g., PDDL-based planners. CX supports a shared world model between a group of agents and provides capabilities for multi-agent task coordination by means of mutual exclusion and resource allocation. The system has been implemented and used successfully on real and simulated robots.

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.

We thank I. Bongartz, M. Gomaa, D. Habering, and T. Viehmann for helpful discussions during CX development.

References

Aha, D. W. 2018. Goal Reasoning: Foundations, Emerging Applications, and Prospects. *AI Magazine* 39(2).

Berry, G., and Gonthier, G. 1992. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming* 19(2).

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. Conference on Automated Planning and Scheduling (ICAPS)*.

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. *6th International Conference on Knowledge Representation and Reasoning (KR)*.

DesJardins, M. E.; Durfee, E. H.; Charles L. Ortiz, J.; and Wolverton, M. J. 1999. A Survey of Research in Distributed, Continual Planning. *AI Magazine* 20(4).

Forgy, C. L. 1982. Rete: A Fast Algorithm for the Many Pattern/-Many Object Pattern Match Problem. *Artificial Intelligence* 19(1).

Hofmann, T.; Niemueller, T.; Claßen, J.; and Lakemeyer, G. 2016. Continual Planning in Golog. In *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).

McGann, C.; Py, F.; Rajan, K.; Thomas, H.; Henthorn, R.; and McEwen, R. 2007. T-REX: A Model-based Architecture for AUV Control. In *3rd Workshop on Planning and Plan Execution for Real-World Systems*.

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.; Ewert, D.; Reuter, S.; Ferrein, A.; Jeschke, S.; and Lakemeyer, G. 2013. RoboCup Logistics League Sponsored by Festo: A Competitive Factory Automation Testbed. In *RoboCup Symposium 2013*.

Niemueller, T.; Reuter, S.; Ferrein, A.; Jeschke, S.; and Lakemeyer, G. 2015. Evaluation of the RoboCup Logistics League and Derived Criteria for Future Competitions. In *RoboCup Symposium 2015 – Development Track*.

Niemueller, T.; Karpas, E.; Vaquero, T.; and Timmons, E. 2016a. Planning Competition for Logistics Robots in Simulation. In *WS on Planning and Robotics (PlanRob) at 26th International Conference on Automated Planning and Scheduling (ICAPS)*.

Niemueller, T.; Zug, S.; Schneider, S.; and Karras, U. 2016b. Knowledge-Based Instrumentation and Control for Competitive Industry-Inspired Robotic Domains. *KI - Künstliche Intelligenz* 30.

Niemueller, T.; Zwilling, F.; Lakemeyer, G.; Löbach, M.; Reuter, S.; Jeschke, S.; and Ferrein, A. 2016c. *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 27th 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.; Hofmann, T.; and Lakemeyer, G. 2018. CLIPS-based Execution for PDDL Planners. In *WS on Integrated Planning, Acting, and Execution (IntEx) at 28th ICAPS*.

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.; Vattam, S.; Alford, R.; Auslander, B.; Karneeb, J.; Molineaux, M.; Apker, T.; Wilson, M.; McMahon, J.; and Aha, D. W. 2014. Iterative Goal Refinement for Robotics. In *WS on Planning and Robotics (PlanRob) at International Conference on Automated Planning and Scheduling (ICAPS)*.

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)*.

Schäpers, B.; Niemueller, T.; Lakemeyer, G.; Gebser, M.; and Schaub, T. 2018. ASP-based Time-Bounded Planning for Logistics Robots. In *28th 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).