28th International Conference on
Automated Planning and Scheduling

June 24-29, 2018, Delft, the Netherlands


# IntEx 2018

Proceedings of the 2nd Workshop on

**Integrated Planning, Acting, and Execution**

**Edited by:**

Tiago Vaquero, Mark Roberts, Sara Bernardini, Tim Niemueller,
and Simone Fratini

## Organization

**Tiago Vaquero**, Jet Propulsion Laboratory, USA

**Mark Roberts**, Naval Research Laboratory, USA

**Sara Bernardini**, Royal Holloway University, London, UK

**Tim Niemueller**, RWTH, Aachen University, Germany

**Simone Fratini**, European Space Agency, Germany

## Program Committee

**Ron Alford**, Mitre Corporation, USA

**Michael Cashmore**, King's College London, UK

**Jeremy Frank**, NASA Ames, USA

**Nir Lipovetzky**, University of Melbourne, Australia

**Dan Magazzeni**, Kings College London, UK

**Fabio Mercorio**, University of MilanBicocca, Italy

**Bob Morris**, NASA Ames, USA

**Tiago Nogueira**, European Space Agency, Germany

**Scott Sanner**, University of Toronto, Canada

**Vikas Shivashankar**, Knexus Research, USA

**Roni Stern**, Ben Gurion University, Israel

**John Winder**, UMBC, USA

# Foreword

Automated planners are increasingly being integrated into online execution systems. The integration may, for example, embed a domain-independent temporal planner in a manufacturing system (e.g., the Xerox printer application) or autonomous vehicles (e.g., a planetary rover or a underwater glider). The integration may resemble something more like a "planning stack" where an automated planner produces an activity or task plan that is further refined before being executed by a reactive controller (e.g., robotics). Or, the integration may be a domain-specific policy that maps states to actions (e.g., reinforcement learning). Models for planning and execution can be integrated or distinct, the planning model can define context-dependent actions schema for on-line (re-)planning or can just specify flexibility to be handled separately at execution time. Online learning may or may not be involved, and may include adjusting or augmenting the model, determining when to repair versus replan, learning to switch policies, etc. A specific focus of these integrations involves online deliberation and execution management, bringing to the foreground concerns over how much computational effort planning should invest over time.

In any of these systems, a planner generates action sequences that are eventually dispatched to an executive, yet taking action in a dynamic world rarely proceeds according to plan. When planning assumptions are challenged during execution, it raises a number of interesting questions about how the system should respond and what is the scope of online deliberation versus execution. Is the "acting" side of the system responsible for a response or the "planning" side? Or do the two need to cooperate and how much? When should the activity planner abandon or preempt the current goals? Should the task planner repair a plan or replan from scratch? Should the executive adjust its current policy, switch to a new one, or learn a new policy from more relevant experience? Objectives and Topics

Similar to IntEx 2017, the workshop aims to (1) provide a forum for discussing the challenges of integrating online planning, acting, and execution, and (2) to assess the potential for holding an integrated execution competitions at ICAPS. We seek original papers concentrating on the following topics:

- online planning, acting, and execution
- position papers, benchmarks, or challenge problems for integrated execution
- improving planning performance from execution experience
- anytime or incremental planning
- discussions of plan dispatching or plan executives
- execution monitoring; comparing replanning, plan repair, regoaling, plan merging
- managing open worlds with closed-world planners; model learning from experience
- determining an observation policy; policy switching; incremental policy adjustment
- modelling, languages and knowledge engineering for interleaved planning and execution
- architectures and application for integrated planning and execution, execution monitoring, mixed-initiative on-line re-planning and execution

Tiago Vaquero, Mark Roberts, Sara Bernardini, Tim Niemueller, and Simone Fratini
June 2018

# Contents

# Automated Adversary Emulation: A Case for Planning and Acting with Unknowns

**Doug Miller, Ron Alford, Andy Applebaum, Henry Foster, Caleb Little, and Blake Strom**
The MITRE Corporation
7515 Colshire Drive
McLean, Virginia 22102
{dpmiller, ralford, aapplebaum, hfoster, clittle, bstrom}@mitre.org

## Abstract

Adversary emulation assessments offer defenders the ability to view their networks from the point of view of an adversary. Because these assessments are time consuming, there has been recent interest in the automated planning community on using planning to create solutions for an automated adversary to follow. We deviate from existing research, and instead argue that automated adversary emulation – as well as automated penetration testing – should be treated as both a planning *and* an acting problem. Our argument hinges on the fact that adversaries typically have to manage unbounded uncertainty during assessments, which many of the prior techniques do not consider. To illustrate this, we provide examples and a formalism of the problem, and discuss shortcomings in existing planning modeling languages when representing this domain. Additionally, we describe our experiences developing solutions to this problem, including our own custom representation and algorithms. Our work helps characterize the nature of problems in this space, and lays important groundwork for future research.

## Introduction

To best understand the security of their systems, network defenders often use *offensive testing* techniques and assessments. These types of assessments come in many forms, ranging from penetration tests – where a team of "white hats" probe the network to identify weaknesses and vulnerabilities – to full-scale red team or even adversary emulation exercises, wherein a team fully emulates an adversary, beginning with reconnaissance, tool and infrastructure development, and initial compromise, and only ending when they reach the specified adversary's goals. As opposed to pure defensive analysis, offensive testing can provide concrete measures of the security of a network by illustrating real attack paths that an adversary could take.

While offensive testing has clear benefits for defenders, it can be difficult for them to actually employ: these tests can be increasingly costly, time-consuming, and personnel constrained. In lieu of easy-to-access offensive testing, an emerging trend in the security community is to launch *automated* offensive assessments. Tools in this space range in capability, from those that focus on technique execution (Smith, Casey 2017) to those that seek to fully emulate an adversary by engaging the full post-compromise adversary life-cycle (Applebaum et al. 2016).

Similarly, the automated planning community has recently taken an interest in security assessments and tests. (Bozic and Wotawa 2017) identify the natural application of automated planning to security: attacks are typically described as a sequence of steps that ultimately achieve a goal, similar in many ways to a plan. They argue that by using automated planning, we can construct tests that we can run against our systems that can identify weaknesses; the authors specifically identify how planning can be used to assess web applications (e.g., SQL injection) as well as the SSL/TLS protocol. Other recent applications include using automated planning and plan recognition to identify larger attack paths (Amos-Binks et al. 2017) as well as vulnerability assessment (Khan and Parkinson 2017).

More specific to offensive testing is the line of work dedicated towards using automated planning specifically for *penetration tests*. Obes, Sarraute, and Richarte (2010) present a model that leverages a deterministic planner alongside a domain description of exploits and connectivity to diagram paths that adversaries could take. Followup work in Sarraute, Buffet, and Hoffmann (2012) expands the model by adding in uncertainty – leaving the core security domain the same – and now using a Partially Observable Markov Decision Process (POMDP) to solve the problem. Shmaryahu et al. (2017) would later acknowledge this POMDP model's success and accuracy, but note its shortcomings – mainly in time-to-compute – as a motivation for using partially observable contingent planning, an approach they argue lies between that of full-knowledge classical planning and multi-belief POMDPs.

Recognizing the wide array of work on automated planning for penetration testing, Hoffman (2015) offers a survey of the literature where he identifies the two main dimensions of existing research: how the approach handles uncertainty from the point of view of the adversary, and how the attack components interact with each other. Hoffman similarly enumerates eight key assumptions, and surveys the literature mapping each to its appropriate assumptions as well as how the approach maps to the two dimensions he identifies.

Despite all of the work dedicated to using automated plan-

ning for penetration testing, little has been done to investigate how the *planning* portion of the problem relates back to the *acting* portion for the problem – most of the approaches assume that the plan will be generated before execution, with the "acting" portion merely following the plan's script. As per Ghallab, Nau, and Traverso (2014), this problem space is not unique in only considering the planning portion of the problem, but for the solutions that have been developed to be commonly deployed, we believe that the field must start embracing acting as part of its paradigm.

**Contribution** In this paper, we argue that automated adversary emulation – and its cousin, automated penetration testing – is not strictly a *planning* problem, but rather a joint *planning and acting problem*. Our argument hinges on a unique characterization of the uncertainty that adversaries face when targeting a network – specifically, that real adversaries work in the face of *unbounded uncertainty*, wherein they are unable to enumerate the outcomes of a sensing action without actually executing it. This makes it all-but-impossible to create an a-priori plan or policy to account for all states, and, accordingly, adversaries that target systems in the wild tend to interleave planning and acting concurrently.

Our views in this paper are influenced heavily by our prior research in (Applebaum et al. 2016) and (Applebaum et al. 2017), as well as our implementation and testing of the CALDERA automated adversary emulation system[1]. As part of this body of work, we consider the task of adversary emulation as opposed to the traditional penetration testing, and in doing so, the specific techniques we consider in this paper are much more varied than those in the literature which focus exclusively on vulnerabilities and exploits. This paper expands on how the adversary emulation problem should be modeled and describes the integrated planning and acting techniques that we have developed to facilitate automated adversary emulation.

## Building Automated Adversary Emulation

The goal of automated adversary emulation is to provide defenders with a tool that is able to execute a full-scale assessment of their network, operating in a way that is similar to a real adversary. Such a tool has significant utility for defenders, including providing a baseline for what their network looks like to an adversary, generating training data, identifying weaknesses and/or misconfigurations, and testing in-place security measures and tools, all the while providing useful empirical evidence for a defensive blue team to build upon. We contrast this with a tool that, for example, only identifies attack paths without executing them: such a tool can provide a map of what the network looks like, but typically will fail to achieve other use cases as it abstracts away important, hard-to-measure details and lacks the realism of actual execution. Specific goals driving automated adversary emulation include:

1. *Intelligent.* The system should choose and chain actions in ways similar to how an adversary would.
2. *Low Overhead.* Defenders should be able to use the tool without needing explicit configuration details, as these are not only time consuming to collect, but are almost impossible for defenders to fully track.
3. *Realism.* The system should execute the same techniques that a real adversary would, and, like a real adversary, should start at initial compromise and only end after achieving (or failing to achieve) a specific set of goals.
4. *Modular.* Users of the system should be able to run assessments with techniques of their choosing, as well as have the ability to add new techniques.

**Adversary Model** In the context of this paper, we use the MITRE ATT&CK framework[2] as our adversary model, specifically focusing on post-compromise techniques – i.e., those used after an adversary has breached a network – that target enterprise systems. ATT&CK provides added insight into the adversary's lifecycle by decomposing it into the top-level tactical goals that adversaries try to achieve and the techniques that adversaries use to achieve those goals. ATT&CK is unlike other threat models in that it was built by analyzing publicly available threat reports – each technique in ATT&CK is grounded in that it has either been used actively by real advanced persistent threats, or that it is common knowledge for red teamers. Moreover, whereas other threat models tend to overly focus on vulnerabilities and exploits, ATT&CK describes behaviors commonly employed by real adversaries, which increasingly involve re-using benign, normal functionality (e.g., built-in system tools) to achieve malicious effects. These features position ATT&CK well within the context of our goals.

## Characterizing Uncertainty in Automated Adversary Emulation

Cyber intrusions can be broken down into a series of constituent actions executed by the adversary. These actions typically fit into two buckets: actions that expand the adversary's foothold, and actions that expand on what the adversary knows. Depending on the circumstances, some actions can span both categories by expanding on the adversary's foothold while also providing new knowledge. To illustrate this, below we describe three common actions – taken from the ATT&CK framework – that adversaries typically execute during engagements.

**Exploiting a Vulnerability** When most people think of cyber attacks, they think of zero-days and exploits used against vulnerable software. With this technique, the adversary expands its foothold by exploiting a vulnerability – i.e., a buffer overflow, remote code inclusion, SQL injection, etc. – on a target system in order to gain access or achieve a malicious effect. Adversaries can have a variety of end goals when using this technique, though common ones include exploiting a remote system for lateral movement and exploiting a local kernel vulnerability for privilege escalation. To successfully launch this technique, an adversary

[1] https://github.com/mitre/caldera

[2] attack.mitre.org

need only have access to the knowledge the vulnerability exists, the right exploit code, and know that the vulnerability is present on the target. This technique expands the adversary's foothold and is an action the adversary takes to acquire new access which can be a new system or level of privilege.

**Remote System Discovery** In order for an adversary to operate against a network, they need to know exactly what systems are on that network. Consider the case for an adversary who just successfully phished an employee within an enterprise; after enumerating the details of the compromised host, they would likely try to seek out another host within the internal network so that they can enlarge their foothold. Before expanding laterally, the adversary would need to *discover* the remote systems first. Depending on the platform, there are many ways to achieve this (e.g., `ping`) – in Windows enterprise systems, the common way is running the `net view` command, which will return a list of all hosts in the local domain. This technique is an example of a *knowledge gaining* technique.

**Credential Dumping** Credential dumping is a favorite for red teamers and real adversaries targeting enterprise systems. To run this technique, an adversary only needs elevated (i.e., SYSTEM) access on a target host. After running it, the technique will extract all cached domain credentials (i.e., passwords and/or hashed passwords) from the running host; cached credentials include all of the credentials of the accounts of users that have logged on since the last reboot. Extracted credentials can be useful later for the adversary as they laterally move through the network, using the stolen credentials to access capabilities they would not otherwise be able to access. This technique both gains new knowledge (e.g., what accounts exist) as well as expands the adversary's territory (i.e., by gaining access to credentials).

Central to each three of these actions is the concept of uncertainty. While not covered here, we note that some actions can also create new domain objects (for example, remotely copying a file from a compromised host to an uncompromised host as a means for lateral movement). This can pose an interesting problem for planning, as many representations do not allow for the creation of new domain objects.

## Managing Uncertainty

Uncertainty factors into each of these techniques in vastly different ways. Consider the first technique, exploiting a vulnerability. In executing this technique, we can characterize two main sources of uncertainty:

- Is the target susceptible to this exploit?
- Was the exploit technique executed successfully?

Both of these questions are *enumerable*: each is a simple yes or no question, making it easy to construct contingency plans that branch over all scenarios. Additionally, these outcomes are *sensible*, in that we can execute other techniques that e.g. check to see if a target is susceptible to an exploit or determine if an exploit ran successfully. Indeed, the approach in (Shmaryahu et al. 2017) explicitly models sensing actions for both. After successfully executing this technique, the adversary will have a new foothold (in the case of lateral movement) or have elevated privileges on an already compromised host (in the case of privilege escalation).

The uncertainty when dumping credentials, however, is different. Unlike exploiting a vulnerability, the uncertainty in dumping credentials is specifically in the *technique output*. When dumping credentials, we know that the adversary will obtain all cached credentials, but in practice the adversary will rarely have any apriori knowledge of what these cached credentials are. This makes it much harder to encode than exploiting a vulnerability, as the adversary may get:

- no credentials;
- credentials for accounts it has never heard of;
- credentials that it can not currently use; or
- credentials that it can immediately use.

In fact, these outcomes tend to occur together: the adversary will likely obtain credentials for accounts it has not heard of while also obtaining credentials for accounts that it has heard of. Enumerating each of these states is possible at the abstract level, but this approach decouples the action from the grounded solution – i.e., references to the objects in the environment, such as John's account or Pete's workstation – making it hard to link the consequences of dumping credentials to enabling actions in the future, and further making it difficult to do goal-based planning to achieve real objectives.

At a more abstract level, adversaries tend to execute techniques that deal with *unbounded uncertainty* when targeting systems; while some of the uncertainty can be characterized, the uncertainty is typically hard to qualify in a way that can be explicitly *bound* without losing too much precision. We contrast this description with *bound uncertainty*. To understand the distinction, consider an action that scans a target for running services to discover exploits. We might consider such an action as being able to identify vulnerability 1, vulnerability 2, ..., etc., where the quantity of vulnerabilities is a known, finite amount that reflects the adversary's toolkit. Scanning for vulnerabilities, then, is a mapping from the unknown state into one where zero or more vulnerabilities – that have already been enumerated beforehand – are known. By contrast, dumping credentials can result in any number of real accounts being discovered, and while each individual account may have some sort of mapping, it is hard for the adversary to explicitly bound the uncertainty, as it does not know the accounts that it does not know.

This kind of scenario is commonplace, and while adversaries will target networks with *specific* goals in mind, they will often bring about those goals in *non-specific* ways. Adversaries typically approach networks with a mental playbook – based on their goals, experience, and results – specifying how they should generally behave, describing their tactical goals as well as the constituent actions they should use throughout the intrusion. How these actions are ordered is left to the adversary to determine at run-time: because of the extreme uncertainty that adversaries have when operating against networks, conformant or contingent plans are too difficult for an adversary to construct when operating in a network. Nonetheless, as we seek to automate the adversary

emulation process, it is critical that we attempt to provide as much detail – and intelligence – that we can in order to have the most accurate results.

## Formal Problem Description

We define our planning problem in two stages: first, the large, agent-agnostic description of the problem, and the smaller adversary-oriented view of what the problem looks like. For the former, we define the general description of the problem as a quadruple: $\pi = \langle \mathcal{P}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{P}$ is a set of propositions, $\mathcal{A}$ a set of actions, $\mathcal{I} \subset P$ the starting state (i.e., a set of propositions), and $\mathcal{G} \subset P$ the goal propositions. For every proposition $p \in \mathcal{P}$, $p$ can be assigned a truth value of either $true$ or $false$. Each action $a \in \mathcal{A}$ is a double, $\{pre_a, post_a\}$, where $pre_a$ is the set of propositions that must be true to execute action $a$ (i.e, the preconditions), and $post_a$ is the set of propositions that will be true after executing $a$. In our model, we assume that the truth of $\mathcal{P}$ is static – i.e., a proposition $p$'s truth value remains the same unless changed by the adversary.

A solution to $\pi$ is a sequence of actions $S = \{a_1, a_2, ..., a_n\}$ such that, when started from $\mathcal{I}$, executing the actions of $S$ in sequence would result in the propositions in $\mathcal{G}$ all being true (or false, if specified as such). We refer to this as an *conformant* solution.

In our scenario, a conformant solution to $\pi$ is unrealistic due to the adversary's uncertainty: because the agent has *unbounded uncertainty*, it is unlikely for our adversary to be able to construct an explicit solution before acting. Thus, we redefine our problem as follows: let $\pi_i$ be a tuple $\pi_i = \langle \mathcal{F}_i, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{F}_i \subseteq \mathcal{P}$ is the set of propositions that the adversary *knows exist* at time $i$, regardless of whether the adversary knows their truth value. At first glance, it might seem that if a proposition $p \in \mathcal{F}_i$, then the adversary knows the truth value of $p$, however this is not always the case. As an example, an adversary may dump credentials and find that $joe$ is a user account. Because $joe$ is a user account, the adversary can infer that $joe$ may be a domain admin – i.e., $domain(joe) \in \mathcal{F}_i$ – but while the adversary can infer this proposition exists, it does not know if it is true.

The differentiator between $\pi$ and $\pi_i$ is in the space of propositions – an actor working with $\pi$ has a fully enumerated proposition space, while an actor working with $\pi_i$ knows that the proposition space is only a subset of what truly exists. Because of this, solving $\pi_i$ is different than solving $\pi$ in that the former *necessitates* acting: the agent *must* perform discovery actions to identify unknown propositions, with planning beyond this point particularly difficult. We can contrast that with an agent working with $\pi$ who knows that there are no new propositions to be gained during an operation, and thus can plan for all contingencies. Under this formalism, then, the adversary emulation problem should not be treated as a strict *planning* problem, but rather as a *selection* problem, where the agent seeks to find the best action *now*, to maximize its chances of achieving $\mathcal{G}$ in the future.

## Representing Planning Problems for Adversary Emulation

The formalism of $\pi_i$ in the previous section calls for the representation of a set of propositions to denote the cyber domain, but is agnostic as to the particular language that represents the propositions. At first glance, this may seem to be a relatively unimportant detail, but our experiments have shown that this situation mandates nuance when represented as a planning problem. Generally, we have observed the following guidelines when tackling this problem:

**Object-oriented description.** Our representations allow us to reason about the objects (i.e., hosts, users, accounts, etc.) typically found in networks in contrast to e.g. state-based approaches. This approach offers many benefits, and is particularly relevant as objects in the cyber domain are well-structured; for example, networks are typically comprised of multiple hosts, each of which have standardized fields such as fully qualified domain name, user accounts that are administrators, operating system, etc. More abstractly, this guideline lends itself well towards *object-oriented planning* (Katz, Moshkovich, and Karpas 2016).

**Parameterized actions.** We tend to refer to our action space as a set of *ungrounded, parameterized* actions; colloquially, we might traditionally refer to these as *techniques*. This, in large part, is due to the uncertainty problem mentioned before: suppose we have a *technique* of dumping credentials from a host. In a traditional representation, we would have $n$ instances of this action represented in $\mathcal{A}$, where $n$ is the number of hosts in the network; i.e., dumping credentials on host 1 is an action, dumping credentials on host 2 is an action, etc. However, because the adversary does not know all of the hosts on the network, it might only have access to a subset of $\mathcal{A}$ alongside the ungrounded, parameterized version of the action.

**Deterministic action outcome.** Executing an action will always result in the same outcome in the same environment. This has likewise been acknowledged in the POMDP approach (Sarraute, Buffet, and Hoffmann 2012), where uncertainty is instead modeled in the adversary's belief space of what the current state is (which indeed more accurately represents what penetration testers do in practice). Alternative approaches, such as the one in (Durkota 2014), abstract this uncertainty instead into the action's outcome.

**Monotonic action consequences.** Our representations are all delete-free. This assumption appears to be consistent with the literature – no representations that we have seen explicitly model deletes – although we note that the solution techniques such as using POMDPs (Sarraute, Buffet, and Hoffmann 2012) or contingency planning (Shmaryahu et al. 2017) are robust enough to handle deletes if the model does include them.

### Early Work in $\mathcal{K}$

Our first attempt (Applebaum et al. 2016), (Applebaum et al. 2017) at modeling this problem was to use a representation encoded in the $\mathcal{K}$ (Eiter et al. 2000) planning language,

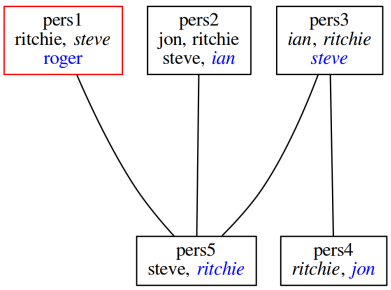



Figure 1: Example encoding of an enterprise system. Each box represents an individual workstation with edges representing allowed traffic flows. The name of the workstation is on the first line within each box, with authorized remote logins in black and local administrators in blue. Active logins are denoted by italics. The red box around $pers1$ signifies the adversary's foothold there.

which we could solve using the DLV$^{K}$[3] planning system. We initially chose $\mathcal{K}$ as the implementation language due to its ability to express uncertainty, our familiarity in Datalog, the ability to encode inferences and axioms, and the availability of the DLV$^{K}$ solver, which itself gave us a fair degree of flexibility during use.

The initial data model that we constructed was simple: it only contained two types of objects, accounts and hosts. Our fluents described both the state of the network – including hosts that were connected and which accounts could log in where – as well as the state of the adversary. This introduced our first challenge, in that we could have a fluent which described the state of the world, and then we would need a corresponding fluent to denote when the adversary was aware of that state. As an example:

```
connected(X, Y) requires host(X), host(Y).
knowsConnected(X, Y) requires host(X), host(Y).
```

Above, the first first predicate `connected(X, Y)` denotes that two hosts in the model can communicate over the network and the second adds it to the adversary's knowledge base. To discover these connections, the adversary has access to a simple action:

```
executable enumerateHost(X) if hasFoothold(X),
   escalated(X), not hostEnumerated(X).
caused knowsConnected(X, Y) if connected(X, Y)
   after enumerateHost(X).
caused hostEnumerated(X) after enumerateHost(X).
```

In words, for the adversary to execute the `enumerateHost` action, it must have an escalated foothold (i.e., executing under root or SYSTEM) on a host and have not previously enumerated it. After execution, it will know all valid connections to or from that host.

Using this problem encoding, we were able to construct plans over our model for the adversary to achieve arbitrary goals. As an example walkthrough, consider the network represented in Figure 1. From this view, we can see a clear path from the adversary's initial foothold on $pers1$ to reach $pers4$ with the following plan: dump credentials on $pers1$ to obtain $steve$'s credentials, use $steve$ to remotely log in to $pers5$, use $steve$ again to move from $pers5$ to $pers3$, dump credentials on $pers3$ to obtain $ritchie$'s credentials, and then use $ritchie$'s account to remotely login to $pers4$[4].

This representation is useful in mapping out the weaknesses in the generated networks from a general perspective, but stops short of being able to entirely represent the features needed for automated adversary emulation: from the adversary's point of view, the adversary only has *partial* view of the network and cannot deterministically reason about the consequences of actions. In Figure 1, the adversary only has a foothold on $pers1$, and without doing anything, only knows that $pers1$ exists, nevermind any of the accounts it would need to laterally move to $pers4$ (or even that $pers4$ exists). Consider the action to dump credentials:

```
executable dumpCreds(X) if hasFoothold(X), escalated(X).
caused knowsCreds(A) if activeCreds(A, X)
    after dumpCreds(X).
```

It is easy to see when the adversary can dump credentials: it only needs an escalated foothold on a host to do so. However, the exact consequences to the adversary are unknown and unbounded: this predicate can only be evaluated *after* running this technique, as the active credentials on a host are unmeasurable from the adversary's point of view.

**Developing a Planning and Acting Environment** To facilitate experiments with the DLV$^{K}$ format, we developed a turn-based simulation system wherein an adversary agent could maintain its own internal state, interfacing with a global agent that had full visibility of the world. Our adversary agent starts with a simple view of the world – it has an initial foothold on the network – as well as the action definitions and inference rules that are used in the real model. It does not, however, know anything about the network: it is unaware of what hosts and accounts exist, what the topology looks like, what the trust relationships are, etc. Instead, the adversary learns these features as it executes actions.

During a simulation run, the adversary sends its chosen action to the global agent, which first checks to see if the action is legal, and, if so, determines what changes should be made to the real model and which new knowledge should be passed to the adversary. As an example, looking at Figure 1, the adversary would gain `knowsRemote(steve, pers1)` and `knowsCreds(steve)` after executing `dumpCreds(pers1)`.

## Pythonic Representation

Our work using $\mathcal{K}$ was useful as a testbed for us to develop planning algorithms. However, from an implementation standpoint, it had several weaknesses, primarily in converting from it to what our tool was executing and what it needed for technique execution; $\mathcal{K}$ did not follow the object-oriented guideline we would ideally follow. This led us to develop a custom representation that easily facilitated both planning and acting.

[3] http://www.dlvsystem.com/k-planning-system/

[4] Note that in our model the adversary would have to perform some knowledge gathering actions throughout this plan as well.

Actions in CALDERA are each represented as a Python class. Each class contains fields that describe how the technique is executed and implemented, some metadata, and then a Python representation of the logic. At a high level, the logic provides information on the pre and postconditions of the actions, represented in a way that talks strictly about the objects that are involved[5]. Treating pre and postconditions as restrictions on objects meshed well with our implementation: the logic explicitly maps to the objects in the database schema and couples well with the action execution code. The internal data model features 15 top level objects, each of which has a set of fields. Field values can either be integers, strings, references to other objects, lists of things, booleans, or dates. Example objects and fields include: Remote Access Trojans (RATs), which have host (object), elevated (boolean), and username (string) fields, and Hosts, which have admins (list), fully qualified domain name (string), and hostname (string) fields, amongst others.

One of the downsides of the Pythonic representation is difficulty for human operators to read. For example, consider the following action to copy a file to a network share:

```
preconditions = [("rat", OPRat),
  ("share", OPShare({"src_host": OPVar("rat.host")}))]
postconditions =
  [("file_g", OPFile({'host': OPVar("share.dest_host")}))]
preproperties = ['rat.executable', 'share.share_path']
postproperties = ['file_g.path']
```

This syntax defines four main components: preconditions, which *must* be true to execute the technique, postconditions, which *at least* will be true after executing the technique, pre-properties, which are things that must be *defined* to execute the technique, and post-properties, which are things that will be defined after executing the technique. Parsing each of these, the first precondition states the the identifier `rat` must be of type `OPRat`. The second requirement states that `share` must be of type `OPShare`, where the `src_host` field is equal to the `rat`'s `host` field. The postcondition states that a new object `file_g` of type `OPFile` will be created, where the `host` field of the new `file_g` object is equal to the original `share`'s `dest_host` field. The pre-properties specify the `rat`'s `executable` and the `share`'s `share_path` fields must also be defined, and the post-properties state that the `file_g`'s `path` field will be defined after execution.

This representation facilitates reasoning over *objects* as opposed to strictly properties, and is handy in the cyber domain: most of the techniques either add knowledge or create objects, with modifying objects an atypical use case. In fact, both adding knowledge and creating objects are represented the same in the Pythonic representation – both involve the agent adding new objects (either those that are discovered or those that are created) to its knowledge base.

**Converting the Pythonic Representation** Instead of reasoning directly with the Pythonic representation, we created an intermediary language that converted the Python requirements to Datalog[6]. For example, the copy action above was converted as follows:

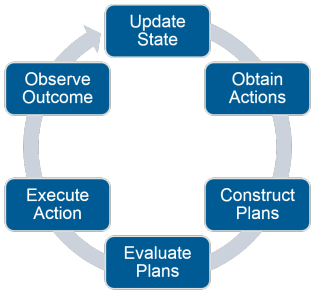



Figure 2: Workflow of plan-and-act paradigms considered for automated adversary emulation.

```
Parameters:
      EXECUTABLE, HOST, RAT, SHARE, SHARE_PATH, SRC_HOST
Preconditions:
      has_property(RAT, executable, EXECUTABLE)
      has_property(RAT, host, SRC_HOST)
      has_property(SHARE, dest_host, HOST)
      has_property(SHARE, share_path, SHARE_PATH)
      has_property(SHARE, src_host, SRC_HOST)
      oprat(RAT)
      opshare(SHARE)
Postconditions:
    + defines_property(FILE_G, path)
    + has_property(FILE_G, host, HOST)
    + opfile(FILE_G)
```

This conversion is relatively straightforward: we see predicates like `oprat(RAT)`, which declare that the `RAT` parameter must be of type `oprat`, matching the Pythonic requirement. Under the preconditions, the second and fifth predicates mandate that the `host` property of `RAT` must be the same as the `src_host` property of `SHARE`. For preproperties, note that the first precondition – `has_property(RAT, executable, EXECUTABLE)` – is the only one to specify a requirement on `EXECUTABLE`; this parameter must merely be defined, but does not need to be explicit.

While the Pythonic code is dense, its Datalog translation is very straightforward. Each object requirement is specified as a type restriction in Datalog. Each preproperty or postproperty is specified as an unbounded `has_property` statement on that object. Preconditions and postconditions are specified with type requirements as well as specific restrictions over properties, again leveraging the `has_property` predicate.

## Choosing Adversary Techniques

In this section, we discuss some of the practical solutions that we have experimented with to solve the planning and acting problem for automated adversary emulation, describing theoretical algorithms that work in our $\mathcal{K}$ environment as

[5]A full description of the syntax can be found at `http://caldera.readthedocs.io/en/latest/add_technique.html`.

[6]For ease of integration, we avoided converting to $\mathcal{K}$ – which required the DLV solver – and instead converted to native Datalog.

well as the one implemented in CALDERA. All of the algorithms discussed in this section leverage the same planning-and-acting paradigm (visualized in Figure 2):

1. Obtain and update the world state.
2. With the world state, use the precondition model to identify which actions are valid at the current time step.
3. Building off of step 2, construct a set of plausible plans.
4. Evaluate each plan constructed in the previous step.
5. Execute the first action in the highest rated plan.
6. Observe the responses, stopping if the goal state has been observed and going back to step 1 otherwise.

This paradigm operates in a way that is *fire-and-forget*: even though the algorithms construct plans, they only execute the first action in the plan, completely re-planning each time they have to construct new plans.

## Evaluating Plans

All of our techniques evaluate plans based on the metric first proposed in (Applebaum et al. 2016). This technique assumes we have access to some reward function $R : \mathcal{A} \rightarrow \mathbb{R}$ that maps each action to some numeric reward. Then, given a set $P$ of plans, where each plan is a sequence of actions $a_1, ..., a_n$: for a plan $p \in P$, we define its score as:

$$S(p) = \sum_{i=1}^{n} \frac{R(a_i)}{i}$$

In words, each plan is assigned a score that represents a decreasing weighted sum over its constituent actions. We note that this is in fact very similar to using a finite horizon over a Markov decision process (MDP) to calculate reward, although here we use a linearly decreasing score as opposed to an exponential one that is typically used in MDPs. In line with the guidelines that we discussed in the previous section, the scoring algorithm treats each action as ungrounded – dumping credentials, for example, on host 1 would yield the same reward as dumping credentials on host 2, regardless of any differences in hosts 1 and 2.

## Constructing Plans

The difficulty in constructing real plans stems primarily from the knowledge disparity facing the adversary: there are propositions in the world-space that the adversary is unaware of. In the generalized case, reasoning over all unknown propositions would be challenging, however, by leveraging our representation – i.e., that we have a data schema, ungrounded action definitions, and a feel for what the world should look like – we can still approximate what might be considered "good" solutions to this problem. Thus, our initial approach worked as follows:

1. Construct a fictional world $\mathcal{P}'$.
2. Merge $\mathcal{P}'$ with $\mathcal{F}_i$. In the case that some proposition in $\mathcal{P}'$ conflicts with $\mathcal{F}_i$, defer to the known proposition to ensure consistency.
3. Initialize an empty set of plans, $P$.
4. For each action type – i.e., for each ungrounded action – obtain the set of plans that executes that action the soonest. For example, if we can dump credentials – regardless of the host – at time step three and no sooner, add all plans of length three that dump credentials to $P$. Repeat this for each action type.
5. Return $P$.

This process is executed multiple times each time step in a Monte-Carlo style simulation: after each individual run, $P$ would be evaluated and the best action would be recorded. After running all of the Monte-Carlo trials, each "best" action would be given a vote, and the action with the most votes would be executed. Experimenting in our $\mathcal{K}$ domain, this setup performed reasonably well, outperforming strategies that iterated through actions in a discrete sequence (i.e., a finite-state machine) as well as a greedy strategy that skipped the plan construction step.

In practice, however, this technique was unfeasible as it required *full world* simulation when constructing $\mathcal{P}'$. This is largely impractical as the more the planner needs to "guess" what the real world looks like, the more inaccuracies it will add. Moreover, creating the entire network and reasoning over it was a time consuming procedure.

To improve on this procedure, we created a variant of the above approach that differed only in how it constructed the initial $\mathcal{P}'$ fictional world. Instead of trying to fully simulate what the world might look like, the planner would make small increments to $\mathcal{F}_i$ based on a fixed set of rules that would enumerate some of the potential configurations. For example, if the planner knew some hosts existed, but did not know the admins on those hosts, the planner would guess who the admins were, with probabilities for guessing that a known or unknown account was an admin. This slight modification provided significant performance increases over our initial approach; the full results of these tests can be found in (Applebaum et al. 2017).

While these two approaches provided strong laboratory-based results, they both sat too far away from the actual CALDERA implementation. Both approaches proved to be too intensive for the large data model that CALDERA was using – how do we, for example, simulate what a random process might look like, given that it has 15 different fields? How many processes should we infer exist? Moreover, the representation in $\mathcal{K}$ did not lend itself well to meeting our guidelines for modeling this domain.

Instead, we developed a relatively simplistic approach that leveraged our Pythonic data representation, converted to Datalog. To get plans, the planner would explore each possible action – in sequence – popping its execution onto a stack and recursing, stopping when it reaches a fixed depth; in essence, the planner explores all possible plans of a fixed length starting at the current state via a depth-first search over the action space. This approach is fairly immature as opposed to traditional planning techniques, however we developed several heuristics to help reduce redundancy and optimize the search:

- The algorithm never explores the same action twice.
- If two actions have the same effects, only explore one.

To understand this, consider the following action to identify all hosts running in a domain:

```
Parameters:
        RAT
Preconditions:
        oprat(RAT)
Postconditions:
      + defines_property(HOST_G, fqdn)
      + defines_property(HOST_G, os_version)
      + ophost(HOST_G)
      + oposversion(OS_VERSION_G)
```

To check if this action is valid, the planner runs a Datalog query to ground its preconditions, in this case returning all objects of type `oprat`. After execution, it will create a new host – "`+ ophost(HOST_G)`" – with the associated properties; from an execution standpoint, the planner, when considering this action, will *internally* execute it, adding the appropriate facts to the knowledge base. In this case, since the postconditions are not tied to the preconditions or parameters, all of the facts will be brand new – that is, it will know that it needs to create new objects to match the conditions. Once it finishes exploring this path, it will pop these postconditions off its stack and move on to the next branch in the tree. To avoid complexity, if multiple RATs match the precondition, it will only explore the paths with one of them since the postconditions are the same, regardless of the RAT. Note that instead of explicitly declaring its in-practice functionality – discovering all hosts – the representation only adds *one* new host to the knowledge base.

Interestingly, this approach completely eschews the need to simulate or guess what the unknown propositions in the world are by leveraging its representation. While we have not conducted any rigorous trials to showcase its efficacy, we have found that, with this algorithm and representation, CALDERA is able to successfully achieve full compromise of setup lab environments. Prior to deploying the techniques in this paper, CALDERA leveraged a hard-coded finite-state machine: by comparison, the planning-based approach is smoother, easier to vary, more adaptable, and much easier to extend, in addition to some efficiency boosts. We note that, because the new approach is much easier to extend and adapt, other researchers were able to use our public implementation of CALDERA to integrate their own techniques – modeling and coding them in our Pythonic representation – with the planner integrating the new actions seamlessly into its operations (Bottomley, P. and Beukema, W. 2018).

## Discussion

Our hope with this paper is twofold: first, to call attention to the need for a planning and acting paradigm within the security and planning community, and second, to raise awareness of the particular types of uncertainty considered in the automated adversary emulation problem. With regards to the latter, our primary call-to-action is inspired by our experiences and design goals: we wish to avoid having users explicitly input network parameters, or even network possibilities, when they run their tests. This contrasts with approaches previously identified in the literature, where they assume access to either a network map, or have clearly bounded uncertainty with which they can run traditional planning techniques. In our own modeling efforts, we have found this to be a difficult task.

We note that the approaches discussed in this paper have been designed to exhibit *emergent* behavior, as opposed to explicit goals or alternative execution strategies. While this is similar to real adversaries, who typically have semi-vague goals (e.g., "exfiltrate all sensitive files"), we believe that this is an area for further research. As it stands, the current heuristic approach prioritizes executing "goal actions" as soon as possible. By contrast, some adversaries may prefer to lay-in-wait, achieving their goals on each host simultaneously – i.e., laterally move to all hosts and then encrypt them, as opposed to encrypting them as you move through the network. Similarly, a system that could achieve a specific goal – i.e., compromise a specific host – would be of great utility for defenders. Towards this, the heuristic approach can be modified to exhibit this type of behavior, however it involves a significant amount of manual analysis of both goal and reward, and is not guaranteed to be optimal. Instead, we believe it may be possible to leverage the unique cyber domain properties to construct a planner that can more accurately achieve this. As it stands, our current approach offers a workable solution for smaller problems, but blows up combinatorially as the depth and domain grows.

Most of the formal modeling in the automated planning community of cyber is either domain specific – i.e., network protocols – or heavily focused on exploits. Because adversaries tend to re-use existing functionality, these latter models lack realism; adversaries do not achieve lateral movement only through exploits. Instead, an ideal representation would cover other ordinarily benign activities that adversaries also use. Based on our experiences, this can be challenging to do from the adversary's perspective (i.e., in bounding the adversary's uncertainty). Additionally, while not covered in high detail in this paper, that adversaries create and reason over new domain objects is a similar encoding issue. We are currently developing a translation from our Pythonic model to PDDL(McDermott et al. 1998), but have found that we often need to use unnatural constructs to represent key cyber concepts. We also plan on investigating the use of epistemic planning (Löwe, Pacuit, and Witzel 2011) to represent the adversary's changing belief states – as encoded, maintaining dual states between the world and the adversary's knowledge is cumbersome, and we believe we can leverage existing strategies to optimize our process.

**Conclusion** Both industry and academia have recognized the utility of automated adversary emulation and penetration testing, the former solving it from an implementation-first perspective, and the latter working on the theory. Neither side, however, seems to have recognized the key challenges that make this a hard problem, nor have others formalized the requirements that an ideal automated offensive solution should meet. We hope that in publishing this paper, we can better characterize these challenges and requirements, helping others better understand the nature of the problem and encouraging future research.

## References


[Amos-Binks et al. 2017] Amos-Binks, A.; Clark, J.; Weston, K.; Winters, M.; and Harfoush, K. 2017. Efficient attack plan recognition using automated planning. In *Computers and Communications (ISCC), 2017 IEEE Symposium on*, 1001–1006. IEEE.

[Applebaum et al. 2016] Applebaum, A.; Miller, D.; Strom, B.; Korban, C.; and Wolf, R. 2016. Intelligent, automated red team emulation. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, 363–373. ACM.

[Applebaum et al. 2017] Applebaum, A.; Miller, D.; Strom, B.; Foster, H.; and Thomas, C. 2017. Analysis of automated adversary emulation techniques. In *Proceedings of the Summer Simulation Multi-Conference*, 16. Society for Computer Simulation International.

[Bottomley, P. and Beukema, W. 2018] Bottomley, P. and Beukema, W. 2018. Signal the ATT&CK: Part 1. `https://www.pwc.co.uk/issues/cyber-security-data-privacy/research/signal-att-and-ck-part-1.html`.

[Bozic and Wotawa 2017] Bozic, J., and Wotawa, F. 2017. Planning the attack! or how to use ai in security testing? In *IWAISe: First International Workshop on Artificial Intelligence in Security*, 50.

[Durkota 2014] Durkota, K. 2014. Computing optimal policies for attack graphs with action failures and costs. In *STAIRS*, 101–110.

[Eiter et al. 2000] Eiter, T.; Faber, W.; Leone, N.; Pfeifer, G.; and Polleres, A. 2000. *Computational Logic — CL 2000: First International Conference London, UK, July 24–28, 2000 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg. chapter Planning under Incomplete Knowledge, 807–821.

[Ghallab, Nau, and Traverso 2014] Ghallab, M.; Nau, D.; and Traverso, P. 2014. The actors view of automated planning and acting: A position paper. *Artificial Intelligence* 208:1–17.

[Hoffmann 2015] Hoffmann, J. 2015. Simulated penetration testing: From "dijkstra" to "turing test++".

[Katz, Moshkovich, and Karpas 2016] Katz, M.; Moshkovich, D.; and Karpas, E. 2016. Lifting delete relaxation heuristics to successor generator planning. *Heuristics and Search for Domain-independent Planning (HSDIP)* 61.

[Khan and Parkinson 2017] Khan, S., and Parkinson, S. 2017. Towards automated vulnerability assessment.

[Löwe, Pacuit, and Witzel 2011] Löwe, B.; Pacuit, E.; and Witzel, A. 2011. DEL planning and some tractable cases. In *International Workshop on Logic, Rationality and Interaction*, 179–192. Springer.

[McDermott et al. 1998] McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. Pddl-the planning domain definition language.

[Obes, Sarraute, and Richarte 2010] Obes, J. L.; Sarraute, C.; and Richarte, G. 2010. Attack planning in the real world. In *Working Notes for the 2010 AAAI Workshop on Intelligent Security (SecArt)*, 10.

[Sarraute, Buffet, and Hoffmann 2012] Sarraute, C.; Buffet, O.; and Hoffmann, J. 2012. Pomdps make better hackers: Accounting for uncertainty in penetration testing. In *Twenty-Sixth AAAI Conference on Artificial Intelligence (AAAI-12)*.

[Shmaryahu et al. 2017] Shmaryahu, D.; Shani, G.; Hoffmann, J.; and Steinmetz, M. 2017. Partially observable contingent planning for penetration testing. In *IWAISe: First International Workshop on Artificial Intelligence in Security*, 33.

[Smith, Casey 2017] Smith, Casey. 2017. Red Canary Introduces Atomic Red Team, a New Testing Framework for Defenders. `https://www.redcanary.com/blog/atomic-red-team-testing/`.

# Using Operational Models to Integrate Acting and Planning

**Sunandita Patra**[1], **Malik Ghallab**[2], **Dana Nau**[3], **Paolo Traverso**[4]
[1,3]University of Maryland, College Park, USA [2]LAAS-CNRS, France, [4]FBK-ICT, Trento, Italy
[1]patras@umd.edu, [2]malik@laas.fr, [3]nau@cs.umd.edu, [4]traverso@fbk.eu

## Abstract

A significant problem for integrating acting and planning is how to maintain consistency between the planner's *descriptive* action models, which abstractly describe *what* the actions do, and the actor's *operational* models, which tell *how* to perform the actions with rich control structures for closed-loop online decision-making. Operational models allow for dealing with a variety of contexts, and responding to unexpected outcomes and events in a dynamically changing environment.

To circumvent the consistency problem, we use the actor's operational models both for acting and for planning. Our acting-and-planning algorithm, APE, uses hierarchical operational models inspired from those in the well-known PRS system. But unlike the reactive PRS algorithm, APE chooses its course of action using a planner that does Monte Carlo sampling over simulated executions of APE's operational models.

Our experiments with this approach show significant benefits in the success rates of the acting system, in particular for domains with dead ends.

## Introduction

The integration of acting and planning is a long-standing AI problem that has been discussed by many authors. For example, (Pollack and Horty 1999) argue that despite successful progress to go beyond the restricted assumptions of classical planning (e.g., handle uncertainty, partial observability, or exogenous events), in most realistic applications just making plans is not enough. Their argument still holds. Planning, as a search over predicted state changes, uses *descriptive models* of actions (*what* might happen). Acting, as an adaptation and reaction to an unfolding context, requires *operational models* of actions (*how* to do things) with rich control structures for closed-loop online decision-making.

A recent survey shows that most approaches to integrating acting and planning seek to combine descriptive and operational representations, using the former for planning and the latter for acting (Ingrand and Ghallab 2017). This has several drawbacks in particular for the development and consistency verification of the models. To ensure consistency, it is highly desirable to have a single representation for both acting and planning. But if this representation were a descriptive one, it wouldn't provide sufficient functionality. Instead, the planner needs to be capable of reasoning directly with the actor's operational models.

In this paper, we provide an integrated acting-and-planning system, APE (Acting and Planning Engine). APE's operational representation language and its acting algorithm are inspired by the well-known PRS system (Ingrand et al. 1996). The operational model is hierarchical: a collection of refinement methods offers alternative ways to handle *tasks* and react to *events*. Each method has a *body* that can be any complex algorithm. In addition to the usual programming constructs, the body may contain *commands* (including sensing commands), which are sent to an execution platform in order to execute them in the real world, and *subtasks*, which need to be refined recursively. APE's acting engine is based on an expressive, general-purpose operational language with rich control structures for closed-loop online decision-making.

To integrate acting and planning, APE extends the reactive PRS-like acting algorithm to include a planner, APE-plan. At each point where APE needs to decide how to refine a task, subtask, or event, APE-plan does Monte Carlo rollouts with a subset of the applicable refinement methods. At each point where a refinement method contains a command to the execution platform, APE-plan takes samples of its possible outcomes using a predictive model of what each command will do.

We have implemented APE and APE-plan and have done preliminary empirical assessments of them on four domains. The results show significant benefits in the success rates of the acting system, in particular for domains with dead ends.

The related work is described in the following section. Then we briefly summarize the operational model. APE and APE-plan are presented in the following section. We present our benchmark domains and experimental results. Finally, we discuss the results and provide conclusions.

## Related Work

To the best of our knowledge, no previous approach has proposed the integration of acting and planning by looking directly within the language of a true operational model like that of APE. Our approach is based on the operational representation language and RAE algorithm in (Ghallab, Nau, and Traverso 2016, Chapter 3), which in turn were inspired

by PRS (Ingrand et al. 1996). RAE operates purely reactively. If it needs to choose among several refinement methods that are eligible for a given task or event, it makes the choice without any attempt to plan ahead. The approach has been extended with some planning capabilities in PropicePlan (Despouys and Ingrand 1999) and SeRPE (Ghallab, Nau, and Traverso 2016). The two systems model commands with classical planning operators; they both require the action models and the refinement methods to satisfy classical planning assumptions of deterministic, fully observable and static environments, which are not acceptable assumptions for most acting systems.

Various acting approaches similar to PRS and APE have been proposed, e.g., (Firby 1987; Simmons 1992; Simmons and Apfelbaum 1998; Beetz and McDermott 1994; Muscettola et al. 1998; Myers 1999); some of these provide refinement capabilities. While such systems offer expressive acting environments, e.g., with real time handling primitives, none of them provide the ability to plan with the operational models used for acting, and thus cannot integrate acting and planning as we propose here. Most of the mentioned systems do not reason about alternative refinements.

Finite State Automata (FSA) and Petri Nets have also been used as representations for acting models, e.g.,(Verma et al. 2005; Wang et al. 1991), again without planning capability. For example, the ROS execution system SMACH (Bohren et al. 2011), implements an automata-based approach, where each state of a hierarchical state machine corresponds to the execution of a command. However, the semantics of constructs available in SMACH is limited for reasoning on goals and states, and there is no planning.

The Reactive Model-based Programming Language (RMPL) (Ingham, Ragno, and Williams 2001) is an object-oriented language that allows a domain to be structured through an object hierarchy with subclasses and multiple inheritance. It combines a system model with a control model, using state-based, procedural control and temporal representations. The system model specifies nominal as well as failure state transitions with hierarchical constraints. The control model uses standard reactive programming constructs. RMPL programs are transformed into Temporal Plan Networks (TPN)(Williams and Abramson 2001), an extension of Simple Temporal Networks with symbolic constraints and decision nodes. Temporal reasoning consists in finding a path, i.e., a plan, in the TPN that meets the constraints. The execution of generated plans allows for online choices (Conrad, Shah, and Williams 2009). TPNs are extended with error recovery, temporal flexibility, and conditional execution based on the state of the world (Effinger, Williams, and Hofmann 2010). Primitive tasks are specified with distributions of their likely durations. A probabilistic sampling algorithm finds an execution guaranteed to succeed with a given probability. Probabilistic TPN are introduced in (Santana and Williams 2014) with the notions of weak and strong consistency. (Levine and Williams 2014) add the notion of uncertainty to TPNs for contingent decisions taken by the environment or another agent. The acting system adapts the execution to observations and predictions based on the plan. RMPL and subsequent developments have been illustrated with a service robot which observes and assists a human. It is a quite comprehensive CSP-based approach for temporal planning and acting; it provides refinement, instantiation, time, nondeterminism, a plan repair. Our approach does not handle time; it focuses instead on decomposition into communicating asynchronous components.

Behavior trees (BT) (Colledanchise 2017; Colledanchise and Ögren 2017) aim at integrating acting and planning within a hierarchical representation. Similarly to our framework, a BT can reactively respond to contingent events that were not predicted. The authors propose a mechanism to synthesize a BT that has a desired behavior. The construction of the tree refines the acting process by mapping the descriptive model of actions onto an operational model. Our approach is different since APE provides the rich and general control constructs of a programming language and we do planning directly within the operational model, rather than through a mapping from the descriptive to an operational model. Moreover, the BT approach does not allow for refinement methods, which are a rather natural and practical way to specify different possible refinements of tasks.

Approaches based on temporal logics or situation calculus (Doherty, Kvarnström, and Heintz 2009; Hähnel, Burgard, and Lakemeyer 1998; Claßen et al. 2012; Ferrein and Lakemeyer 2008) specify acting and planning knowledge through high-level descriptive models and not through operational models like used in APE. Moreover, these approaches integrate acting and planning without exploiting the hierarchical approach based on refinement methods described in this paper.

Our framework has some similarities with HTN (see, e.g., (Nau et al. 1999)), since tasks can be refined with different methods. However, our methods are significantly different from HTN ones since our methods are programs that can encode rich control constructs rather than simple sequences of primitive tasks. This is what allows us to provide a framework for acting and planning.

(Bucchiarone et al. 2013) propose a hierarchical representation framework that includes abstract actions and that can interleave acting and planning for composing web services. However this work focus on distributed processes, which are represented as state transition systems, and does not allow for refinement methods.

Finally, a wide literature on probabilistic planning and Monte Carlo tree search refers to simulated execution, e.g., (Feldman and Domshlak 2013; Feldman and Domshlak 2014; Kocsis and Szepesvári 2006; James, Konidaris, and Rosman 2017) and sampling outcomes of action models e.g., the RFF algorithm (Teichteil-Königsbuch, Infantes, and Kuter 2008), FF-replan (Yoon, Fern, and Givan 2007) and hindsight optimization (Yoon et al. 2008). Beyond the fact that all these works are based on a probabilistic MDP framework, the main conceptual and practical difference with our work is that they consider just a descriptive model, i.e., abstract actions on finite MDPs. Their focus is therefore entirely on planning, and do not allow for an integration of acting and planning. Most of the papers refer to doing the planning online – but they are doing the planning using descriptive models rather than operational models. There is no

notion of integration of acting and planning, hence no notion of how to maintain consistency between the planner's descriptive models and the actor's operational models. Moreover, they have no notion of hierarchy and refinement methods.

## Operational Models

Our formalism for operational models of actions is based on the one in (Ghallab, Nau, and Traverso 2016, Chapter 3). It has features that allow for dealing with a dynamic environment which has other actors and exogenous events. The main ingredients are *tasks*, *events*, *commands*, *refinement methods*, and *state variables*. Some of the state variables are *observable*, i.e., the execution platform will automatically keep them up-to-date through sensing operations. We illustrate this representation through the following examples.

**Example 1.** *Consider several robots (UGVs and UAVs) moving around in a partially known terrain, performing operations such as data gathering, processing, screening and monitoring. In this domain, let*

- $R = \{g_1, g_2, a_1, a_2\}$ *be the set of robots,*
- $L = \{base, z_1, z_2, z_3, z_4\}$ *be the set of locations,*
- survey$(r, l)$ *be a command performed by robot $r$ in location $l$ that surveys $l$ and collects data*
- loc$(r) \in L$ *and* data$(r) \in [0, 100]$ *be observable state variables that contain the robot $r$'s current location and the amount of data it has collected.*

*Let* explore$(r, l)$ *be a task for robot $r \in R$ to reach location $l \in L$ and perform the command* survey$(r, l)$*. In order to survey, the robot needs some equipment that might either be available or in use by another robot. Robot $r$ should collect the equipment, then move to the location $l$ and execute the command* survey$(r, l)$*. Each robot can carry only a limited amount of data. Once its data storage is full, it can either go and deposit data to the base, or transfer it to an UAV via the task* depositData$(r)$*. Here is a refinement method to do this.*

```
m1-explore(r, l)
  task: explore(r, l)
  body: get-Equipment(r, 'survey')
        moveTo(r, l)
        if loc(r) = l then:
            Execute command survey(r, l)
            if data(r) = 100 then:
                depositData(r)
            return success
        else return failure
```

*Above,* get-Equipment$(r,$ *'survey'*$)$*,* moveTo$(r, l)$ *and* depositData$(r)$ *are subtasks which need to be further refined via suitable refinement methods. Only UAVs have the ability to fly. So, there can be different possible refinement methods for the task* moveTo$(r, l)$ *based on whether $r$ can fly or not.*

*Each robot can hold a limited amount of charge and is rechargeable. Depending on what locations it needs to survey, it might need to recharge by going to the base where the charger is located. Different ways of doing this can be captured by multiple refinement methods for the task* doActivities$(r, locList)$*. Here are two of them:*

```
m1-doActivities(r, locList)
  task: doActivities(r, locList)
  body: for l in locList do:
            explore(r, l)
        moveTo(r, 'base')
        if loc(r) = 'base':
            recharge(r)
        else return failure
        return success
```

```
m2-doActivities(r, locList)
  task: doActivities(r, locList)
  body: for l in locList do:
            explore(r, l)
        moveTo(r, 'base')
        if loc(r) = 'base':
            recharge(r)
        else return failure
        return success
```

Note that a refinement method for a task $t$ specifies *how to* perform $t$, i.e., it gives a procedure for accomplishing $t$ by performing subtasks, commands and state variable assignments. This procedure can include any of the usual programming constructs, e.g., if-then-else, loops and so forth.

The above example illustrates tasks and refinement methods. Let us give the robots a method for reacting to an event.

**Example 2.** *Suppose that an alien is spotted in one of the locations $l \in L$ of Example 1 and a robot has to react to it by stopping its current activity and going to $l$. Let us represent this with an event* alienSpotted$(l)$*. We also need an additional state variable:* alien-handling$(r) \in \{$T, F$\}$ *which indicates whether the robot $r$ is engaged in handling an alien. Here is a refinement method for this event:*

```
m-handleAlien(r, l)
  event: alienSpotted(l)
  body: if alien-handling(r) = F then:
            alien-handling(r) ← T
            moveToAlien(r, l)
            Execute command negotiate-with-alien(r, l)
            alien-handling(r) ← F
            return success
        else return failure
```

*This method can succeed if robot $r$ is not already engaged in negotiating with another alien. After negotiations are over, the methods changes the value of* alien-handling$(r)$ *to F.*

## APE and APE-plan

Algorithm 1, APE (Acting and Planning Engine), is based loosely on the RAE (Refinement Acting Engine) algorithm in (Ghallab, Nau, and Traverso 2016, Chapter 3). APE's first inner loop (line (1)) reads each new *job*, i.e., each task or event that comes in from an *external source* such as the user or the execution platform, as opposed to the subtasks generated by APE's refinement methods. For each such job $\tau$, APE creates a *refinement stack* analogous to a computer program's execution stack. *Agenda* is the set of all current refinement stacks.

In the second inner loop (line (4)), for each refinement stack in *Agenda*, APE *progresses* the topmost *stack element* by one step. The stack element includes (among other things) a task or event $\tau$ and the method instance $m$ that APE has chosen to use for $\tau$. The body of $m$ is a program, and progressing the stack element (the Progress subroutine) means executing the next step in this program. This may involve monitoring the status of a currently executing command (line (6)), following a control structure such as a loop or if-then-else (line (7)), executing an assignment statement, sending a command to the execution platform, or handling a subtask $\tau'$ by pushing a new stack element onto the stack

```
APE( )
  Agenda ← empty list
  loop
    for each new task or event τ in the input stream, do                (1)
      s ← current state
      M ← {applicable method instances for τ in state s}
      T ← APE-plan(M, s, τ)                                             (2)
      if T = failed then output("failed to address", τ)
      else do
        m ← the method instance at the top of T                         (3)
        stack ← a new, empty refinement stack
        push (τ, m, nil, ∅) onto stack
        insert stack into Agenda
    for each stack ∈ Agenda do                                          (4)
      Progress(stack)
      if stack is empty then remove it from Agenda                      (5)

Progress(stack)
  (τ, m, step, tried) ← top(stack)
  if step ≠ nil then  // i.e., if we have started executing m
    // step is the current step of m
    if type(step) = command then                                        (6)
      // step is running on the execution platform
      case execution-status(step):
        still-running: return
        failed:        Retry(stack); return
        successful:    pass     // continue to next line
  if there are no more steps in m then pop(stack); return
  step ← next step of m after accounting for the effects
         of control statements (loops, if-then-else, etc.)              (7)
  case type(step):
    assignment: update s according to step; return
    command:
        send step to the execution platform; return                     (8)
    task: pass          // continue to next line
  τ' ← step;  s ← current state                                         (9)
  M' ← {applicable method instances for τ' in state s}
  T' ← APE-plan(M', s, τ')                                              (10)
  if T' = failed then Retry(stack); return
  m' ← the method instance at the top of T'                             (11)
  push (τ', m', nil, ∅) onto stack                                      (12)

Retry(stack)
  (τ, m, step, tried) ← pop(stack)
  add m to tried   // the things we tried that didn't work
  s ← current state
  M ← {applicable method instances for τ in state s}
  T ← APE-plan(M \ tried, s, τ)                                         (13)
  if T ≠ failed then
    m' ← the method instance at the top of T                            (14)
    push (τ, m', nil, tried) onto stack
  else if stack is empty then
    output("failed to accomplish", τ)
    remove stack from Agenda
  else Retry(stack)
```

Algorithm 1: APE, the Acting and Planning Engine.

(line (12)). A method succeeds in accomplishing a task when it returns without any failure.

Whenever APE creates a stack element for a task $\tau$, it must choose (lines (3), (11), and (14)) a method instance $m$ for $\tau$. In order to make an informed choice of $m$, APE calls (lines (2), (10), and (13)) a planner, APE-plan, that returns a plan for accomplishing $\tau$. The returned plan, $T$, will begin with a method $m$ to use for $\tau$. If $m$ contains subtasks, then $T$ must include methods for accomplishing them (and so forth recursively), so $T$ is a tree with $m$ at the root.

Once APE has selected $m$, it ignores the rest of $T$. Thus in line (9), where $m$ has a subtask $\tau'$, APE doesn't use the method that $T$ used for $\tau'$. Instead, in line (11), APE calls APE-plan to get a new plan $T'$ for $\tau'$. This is a receding-horizon search analogous to how a game-playing program might call an alpha-beta game-tree search at every move.[1]

The pseudocode of APE-plan is given in the appendix. It is a modified version of the APE pseudocode that incorporates these main modifications:

1. Each call to APE-plan returns a *refinement tree* $T$ whose root node contains a method instance $m$ to use for $\tau$. The children of this node include a refinement tree (or terminal node) for each subtask (or command, respectively) that APE-plan produced during its Monte Carlo rollout of $m$.
2. In lines (2), (10), and (13), APE-plan calls itself recursively on a set $M' \subseteq M$ that contains the first $b$ members of $M$ a list of method instances ordered according to some domain-specific preference order (with $M' = M$ if $|M| < b$), where $b$ is a parameter called the *search breadth*.This produces a set of refinement trees. If the set is nonempty, then APE-plan chooses one that optimizes cost, time or any other user-specified objective function. If the set is empty, then APE-plan returns the first method instance from $M'$ if $|M'| >= 1$; otherwise it returns failed. See Figures 5 and 6 in the appendix for more details.
3. Each call to Retry is replaced with an expression that just returns failed. While APE needs to retry in the real world with respect to the real actual state, APE-plan considers that a failure is simply a dead end for that particular sequence of choices.
4. In line (8) (the case where step is a command), instead of sending *step* to the actor's execution platform, APE-plan invokes a predictive model of what the execution platform would do. Such a predictive model may be any piece of code capable of making such a prediction, e.g., a deterministic, nondeterministic, or probabilistic state-transition model, or a simulator of some kind. Since different calls to the predictive model may produce different results, APE-plan calls it $b'$ times, where $b'$ is a parameter called the *sample breadth*. From the $b'$ trial runs, APE-plan gets an estimate of *step*'s expected time, cost, and probability of leading to success. See Figures 8 and 10 in the appendix for more details.

[1](Ghallab, Nau, and Traverso 2016) describes a "lazy lookahead" in which an actor keeps using its current plan until an unexpected outcome or event makes the plan incorrect, and a "concurrent lookahead" in which the acting and planning procedures run concurrently. We tried implementing these for APE, but in our experimental domains they did not make much difference in APE's performance.

5. Finally, APE-plan has a *search depth* parameter $d$. When APE calls APE-plan, APE-plan continues planning either to completion or depth $d$, whichever comes earlier. Such a parameter can be useful in real-time environments where there may not be enough time to plan all the way to completion.

## Experimental Evaluation

### Domains

We have implemented and tested our framework on four domains. The Explorable Environment domain (EE) extends the UAVs and UGVs setting of Example 1 with some additional tasks and refinement methods. This domain has dead ends because a robot may run of charge in an isolated location.

The Chargeable Robot Domain (CR) consists of several robots moving around to collect objects of interest. The robots can hold a limited amount of charge and are rechargeable. To move from one location to another, the robots use Dijkstra's shortest path algorithm. The robots don't know where objects are unless a sensing action is performed in the object's location. They have to search for an object before collecting it. Also, the robot may or may not carry the charger with it. The environment is dynamic due to emergency events as in Example 2. A task reaches a dead end when a robot, which is far away from the charger, has run out of charge.

The Spring Door domain (SD) has several robots are trying to move objects from one room to another in an environment with a mixture of spring doors and ordinary doors. Spring doors close themselves unless they are held. A robot cannot carry an object and hold a door simultaneously. So, whenever it needs to move through a spring door, it needs to ask for help from another robot. Any robot which is free can act as the helper. The environment is dynamic because the the type of door is unknown to the robot. But, there are no dead ends.

The Industrial Plant domain (IP) consists of an industrial workshop environment, as in the RoboCup Logistics League competition. There are several fixed machines for painting, assembly, wrapping and packing. As new orders for assembly, paint, etc., arrive, carrier robots transport the necessary objects to the required machine's location. An order can be complex, like, paint two objects, assemble them together, and pack the resulting object. Once the order is done, the final product is delivered to the output buffer. The environment is dynamic because the machines may get damaged and need repair before being used again; but there are no dead ends.

These four domains have different properties, summarized in Figure 1. CR includes a model for the sensing action where the robot can sense a location and identify objects in that location. SD models a situation where robots need to collaborate with each other. They can ask for help from each other. EE models a combination of robots with different capabilities (UGVs and UAVs) whereas in the other three domains all robots have same capabilities. It also models collaboration like the SD domain. In the IP domain, the allocation of tasks among the robots is hidden from the user. The user just specifies their orders; the delegation of the subtasks (movement of objects to the required locations) is handled inside the refinement methods. CR and EE are domains that can represent dead-ends, whereas SD and IP do not have dead-ends.

| Domain | Dynamic events | Dead ends | Sensing | Robot collaboration | Concurrent tasks |
|---|---|---|---|---|---|
| CR | ✓ | ✓ | ✓ | – | ✓ |
| EE | ✓ | ✓ | – | ✓ | ✓ |
| SD | ✓ | – | – | ✓ | ✓ |
| IP | ✓ | – | – | ✓ | ✓ |

Figure 1: Properties of our domains

### Experiments and Analysis

The objective of our experiments was to examine how APE's performance might depend on the amount of planning that we told APE to do. For this purpose, we created a suite of test problems. Each test problem included 1 to 4 jobs to accomplish, and for each job, there was a randomly chosen time point at which it would arrive in APE's input stream.

The amount of planning done by APE-plan depends on its search breadth $b$, sample breadth $b'$, and search depth $d$. We used $b' = 1$ (one outcome for each command), and $d = \infty$ (planning always proceeded to completion), and five different search breadths, $b = 0, 1, 2, 3, 4$. Since APE tries $b$ alternative refinement methods for each task or subtask, the number of alternative plans examined by APE is exponential in $b$. As a special case, $b = 0$ means running APE in a purely reactive way without any planning at all. Our objective function for the experiments is the number of commands in the plan.

In the CR, EE, SD and IP domains, our test suites consisted of 60, 54, 60, and 84 problems, with the numbers of jobs to accomplish being 114, 126, 84 and 276, respectively. In our experiments we used simulated versions of the four environments, running on a 2.6 GHz Intel Core i5 processor.

**Success ratio.** Figure 2 shows APE's *success ratio*, the proportion of jobs that it successfully accomplished in each domain. For the two domains with dead ends (CR and EE), the success ratio generally increases as we increase the value of $b$. In the CR domain, the success ratio makes a big jump from $b = 1$ to $b = 2$ and then remains nearly the same for $b = 2, 3, 4$. This is because for most of the CR tasks, the second method in the preference ordering (in our experiments, this order is decided by the domains' author)turned out to be the best one, so higher value of $b$ did not help much. In contrast, in the EE domain, the success ratio continued to improve significantly for $b = 3$ and $b = 4$.

In the domains with no dead ends, $b$ didn't make very much difference in the success ratio. In the IP domain, $b$ made almost no difference at all. In the SD domain, the success ratio even decreased slightly from $b = 1$ to $b = 4$. This is because in our preference ordering for the tasks of the SD domain, the methods appearing earlier are better suited to handle the events in our problems whereas the methods ap-

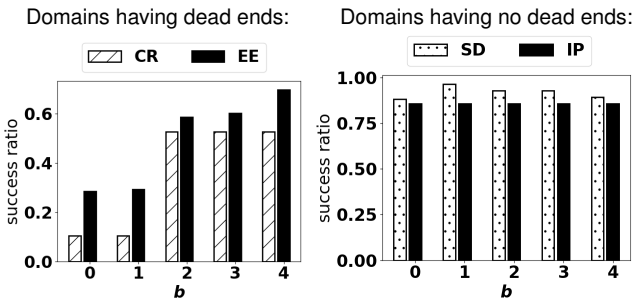



Figure 2: Success ratio (number of successful jobs / total number of jobs) for different values of search breadth $b$.

pearing later produce plans that are shorter but less robust to unexpected events. These experiments confirm the expectation that planning is critical in domains where the actor may get stuck in dead ends. It also has benefits in acting costs (the *retry ratio* and *speed to success* measurements described below).

**Retry ratio.** Figure 3 shows the *retry ratio*, i.e., the number of times that APE had to call the Retry procedure, divided by the total number of jobs to accomplish. The Retry procedure is called when there is a failure in the method instance $m$ that APE chose for some task $\tau$ (see Algorithm 1). Retry works by trying to use another applicable method instance for $\tau$ that it hasn't tried already. Although this is a little like backtracking, a critical difference is that since the method $m$ has already been partially executed, it has changed the current state, and in real-world execution (unlike planning), there is no way to backtrack to a previous state. In many application domains it is important to minimize the total number of retries, since recovery from failure may incur significant, unbudgeted amounts of time and expense.

In all four of the domains, the retry ratio decreases slightly from $b = 0$ (purely reactive APE) to $b = 1$, and it generally decreases more as $b$ increases. This is because higher values of $b$ make APE-plan examine a larger number of alternative plans before choosing one, thus increasing the chance that it finds a better method for each task. In the CR domain, the big decrease in retry ratio from $b = 1$ to $b = 2$ corresponds to the increase in success ratio observed in Figure 2. The same is true for the EE domain at $b = 2$ and $b = 4$. Since the retry ratio decreases with increasing $b$ in all four domains, this means that the integration of acting and planning in APE is important in order to reduce the number of retries.

**Speed to success.** An acting-and-planning system's performance cannot be measured only with respect to the time to plan; it must also include the *time to success*, i.e., the total amount of time required for both planning and acting. Acting is in general much more expensive, resource demanding, and time consuming than planning; and unexpected outcomes and events may necessitate additional acting and planning.

For a successful job, the time to success is finite, but for a failed job it is effectively infinite. To make all of the numbers finite so that they can be averaged, we use the reciprocal

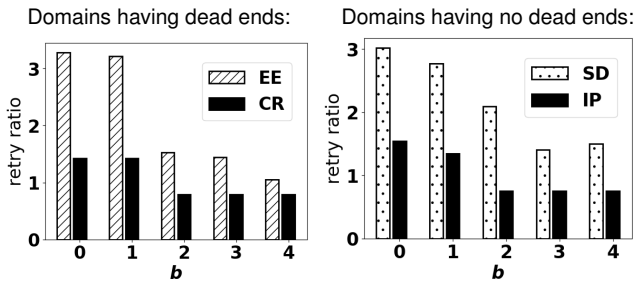



Figure 3: Retry ratio (number of retries / total number of jobs) for different values of search breadth $b$.

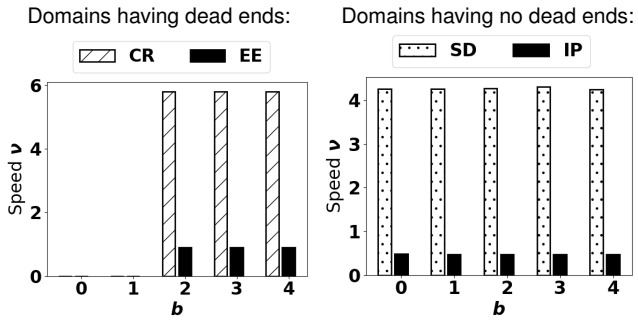



Figure 4: Speed to success $\nu$ averaged over all of the jobs, for different values of search breadth $b$.

amount, the *speed to success*, which we define as follows:

$$\nu = \begin{cases} 0 & \text{if the job isn't successful,} \\ \alpha/(t_p + t_a + n_c t_c) & \text{if the job is successful,} \end{cases}$$

where $\alpha$ is a scaling factor (we used $\alpha = 10,000$, otherwise all of our numbers would be very small), $t_p$ and $t_a$ are APE-plan's and APE's total computation time, $n_c$ is the number of commands sent to the execution platform, and $t_c$ the average amount of time needed to perform a command. In our experiments we used $t_c = 250$ seconds. The higher the average value of $\nu$, the better the performance.

Figure 4 shows how the average value of $\nu$ depends on $b$. In the domains with dead-ends (CR and EE), there is a huge improvement in $\nu$ from $b = 1$ (where $\nu$ is nearly 0) to $b = 2$. This corresponds to a larger number of successful jobs in less time. As we increase $b$ further, we only see slight change in $\nu$ for all the domains even though the success ratio and retry ratio improve (Figures 2 and 3). This is because of the extra time overhead of running APE-plan with higher $b$.

In summary, for domains with dead ends, planning with APE-plan outperforms purely reactive APE. The same occurs to some extent in the domains without dead ends, but there the effect is less pronounced thanks to the good domain specific heuristics in our experiments.

## Concluding Remarks

We have proposed a novel algorithm APE for integrating acting and planning using the actor's operational models.

Our experimentation covers different interesting aspects of realistic domains, like dynamicity, and the need for run-time sensing, information gathering, collaborative and concurrent tasks (see Figure 1). We have shown the difference between domains with dead ends, and domains without dead ends through three different performance metrics: the success ratio, retry ratio and speed to success. We saw that acting purely reactively in the domains with dead ends can be costly and dangerous. The homogenous and sound integration of acting and planning provided by APE is of great benefit for domains with dead ends which is reflected through a higher success ratio. In most of the cases, the success ratio increases with increase in the parameter, search breadth, $b$ of APE-plan. In the case of safely explorable domains, APE manages to have a similar ratio of success for all values of $b$.

Our second measure, the retry ratio, counts the number of retries of the same task done by APE before succeeding. Performing many retries is not desirable, since this has a high cost and faces the uncertainty of execution. We have shown that both in domains with dead ends and without, the retry ratio significantly diminishes with APE-plan, demonstrating the benefits of using APE-plan also in safely explorable domains.

Finally we have devised a novel, and we believe realistic and practical way, to measure the performance of APE and similar systems. While most often the experimental evaluation of systems addressing acting and planning is simply performed on the sole planning functionality, we devised a *speed to success* measure to assess the overall time to plan and act, including failure cases. It takes into account that the time to execute commands in the real world are usually much longer than the actor's computation time. We have shown that, in general, the integration of APE-plan reduces time significantly in the case of domains with dead ends, while there is not such significant decrease in performance in the case of safely explorable domains.

Future work will include more elaborate experiments, with more domains and test cases, and different settings of APE-plan's search breadth, search depth, and sample breadth parameters. We also plan to test with different heuristics, compare APE with other approaches cited in the related work, and finally do testing in the physical world with actual robots.

## Appendix

In this section, we describe the pseudocode of APE-plan, the planner of our acting-and-planning engine, APE. $b$, $b'$ and $d$ are global variables representing the search breadth, sample breadth and search depth respectively (described in the main paper). The main procedure of APE-plan is shown in Figure 5. APE-plan receives as input a task $\tau$ to be planned for, a set of methods $M$ and the current state $s$. APE-plan returns a refinement tree $T$ for $\tau$. It starts by creating a refinement tree with a single node $n$ labeled $\tau$ and calls a sub-routine APE-plan-Task which builds a complete refinement tree for $n$.

APE-plan has three main sub-procedures: APE-plan-Task, APE-plan-Method and APE-plan-Command. APE-plan-Task looks at $b$ method instances for refining a task $\tau$. It calls

```
APE-plan (M, s, τ)
  n ← new tree node
  label(n) ← τ
  T_0 ← tree with only one node n
  (T, v) ← APE-plan-Task(s, T_0, n, M, 0)
  if v ≠ failure then
    return (T, v)
  else:
    B ← { Applicable method instances for τ in M
          ordered according to a preference ordering }
    if B ≠ ∅ then
      n ← Create new node
      label(n) ← B[1]
      T ← tree with only one node n as the root
      return (T, 0)
    else:
      return null, failure
```

Figure 5: The pseudocode of the planner used by APE

APE-plan-Method for each of the $b$ method instances and returns the tree with the most optimal *value*. Every refinement tree has a value based on probability and cost. Once APE-plan-Task has chosen a method instance $m$ for $\tau$, it re-labels the node $n$ from $\tau$ to $m$, in the current refinement tree $T$. Then it simulates the steps in $m$ one by one by calling the sub-routine APE-plan-Method.

```
APE-plan-Task (s, T, n, M, d_curr)
  τ ← label(n)
  B ← { Applicable method instances for τ in M ordered
        according to a preference ordering }
  if |B| < b then
    B' ← B
  else:
    B' ← B[1...b]
  U, V ← empty dictionaries
  for each m ∈ B' do
    label(n) ← m
    U[m], V[m] ← APE-plan-Method(
                          s, T, n, M, d_curr + 1)
  m_opt ← arg-optimal_m {V[m]}
  return (U[m_opt], V[m_opt])
```

Figure 6: The pseudocode for APE-plan-Task

APE-plan-Method first checks whether the search has reached the maximum depth. If it has reached the maximum depth, APE-plan-Method makes an heuristic estimate of the cost and predicts the next state after going through the steps present inside the method. Otherwise, it creates a new node in the current refinement tree $T$ labeled with the first step in the method. If the step is a task, then APE-plan-Task is called for the task. If the step is a command, then APE-plan-Method calls the sub-routine APE-plan-Command.

APE-plan-Command first calls the sub-routine SampleCommandOutcomes. SampleCommandOutcomes samples $b'$ outcomes of the command *com* in the current state $s$. The sampling is done from a probability distribution specified by the domain's author. SampleCommandOutcomes re-

```
APE-plan-Method (s, T, n, M, d_curr)
  m ← label(n)
  if d_curr = d then
      s', cost' ← HeuristicEstimate(s, m)
      n', d' ← NextStep (s', T, n, d_curr)
  else:
      step ← first step in m
      n' ← new tree node
      label(n') ← step
      Add n' as a child of n
      d' ← d_curr
      cost' ← 0
      s' ← s
  case type(label(n')):
      task: T', v' ← APE-plan-Task(s', T, n', M, d')
      command: T', v' ← APE-plan-Command(
                                    s', T, n', M, d')
      end: T' ← T; v' ← 0
  return (T', v' + cost')
```

Figure 7: The pseudocode for APE-plan-Method

```
APE-plan-Command (s, T, n, M, d_curr)
  res ← SampleCommandOutcomes (s, label(n))
  value ← 0
  for (s', v, p) in res do
      n', d' ← NextStep (s', T, n, d_curr)
      case type(label(n')):
          command:
              T_s', v_s' ← APE-plan-Command(
                                    s', T, n', M, d_curr)
          task:
              T_s', v_s' ← APE-plan-Task(s', T, n', M, d_curr)
          end:
              T_s' ← T; v_s' ← 0
      value ← value + (p * (v + v_s'))
  return T, value
```

Figure 8: The pseudocode for APE-plan-Command

turns a set consisting of three tuples of the form $(s', v, p)$, where $s'$ is a predicted state after performing command *com*, and $v$ and $p$ are the cost and probabilities of reaching that state estimated from the sampling. We need the next state $s'$ to build the remaining portion of the refinement tree $T$ starting from the state $s'$. The cost $v$ contributes to the expected value of $T$ with probability $p$. Now, after getting this list of three tuples from SampleCommandOutcomes, APE-plan-Command calls the NextStep sub-routine.

NextStep (shown in Figure 9) takes as input the current refinement tree $T$ and the current node $n$ being explored. If $n$ refers to some task or command in the middle of a refinement method $m$, then NextStep creates a new node labeled with the next step inside $m$. The depth of $n_{next}$ will be same as $n$. Otherwise, if $n$ is the last step of $m$, it continues to loop and travel towards the root of the refinement tree until it finds the root or a method that has not been fully simulated yet. It returns `end` when $T$ is completely refined or a node labeled with the next step in $T$ according to $s$ and its depth.

```
NextStep (s, T, n, d_curr)
  d_next ← d_curr
  while(True)
      n_old ← n
      n ← parent(n_old) in T
      m ← label(n)
      step ← next step in m after label(n_old) depending on s
      if step is not the last step of m then
          n_next ← new tree node
          label(n_next) ← step
          break
      else
          d_next ← d_next − 1
          if d_next = 0 then
              n_next ← new tree node
              label(n_next) ← end
              break
          else
              continue
  return n_next, d_next
```

Figure 9: The sub-routine NextStep

After APE-plan-Command gets a new node $n'$ and its depth from NextStep, it calls APE-plan-Command or APE-plan-Task depending on the label of $n'$. It does this for every $s'$ in *res* and estimates a value for $T$ from these runs.

```
SampleCommandOutcomes (s, com)
  S ← φ
  Cost, Count ← empty dictionaries
  loop b' times:
      s' ← Sample(s, com)
      S ← S ∪ {s'}
      if s' in Count:
          Count[s'] ← 1
          Cost[s'] ← cost_{s,m[i]}(s')
      else:
          Count[s'] ← Count[s'] + 1
  normalize(Count)
  res ← φ
  for s' ∈ S do
      res ← res ∪ {(s', Cost[s'], Count[s'])}
  return res
```

Figure 10: The sub-routine SampleCommandOutcomes

## References


[Beetz and McDermott 1994] Beetz, M., and McDermott, D. 1994. Improving robot plans during their execution. In *AIPS*.

[Bohren et al. 2011] Bohren, J.; Rusu, R. B.; Jones, E. G.; Marder-Eppstein, E.; Pantofaru, C.; Wise, M.; Mösenlechner, L.; Meeussen, W.; and Holzer, S. 2011. Towards autonomous robotic butlers: Lessons learned with the PR2. In *ICRA*, 5568–5575.

[Bucchiarone et al. 2013] Bucchiarone, A.; Marconi, A.; Pistore, M.; Traverso, P.; Bertoli, P.; and Kazhamiakin, R. 2013.

Domain objects for continuous context-aware adaptation of service-based systems. In *ICWS*, 571–578.

[Claßen et al. 2012] Claßen, J.; Röger, G.; Lakemeyer, G.; and Nebel, B. 2012. Platas—integrating planning and the action language Golog. *KI-Künstliche Intelligenz* 26(1):61–67.

[Colledanchise and Ögren 2017] Colledanchise, M., and Ögren, P. 2017. How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees. *IEEE Trans. Robotics* 33(2):372–389.

[Colledanchise 2017] Colledanchise, M. 2017. *Behavior Trees in Robotics*. Ph.D. Dissertation, KTH, Stockholm, Sweden.

[Conrad, Shah, and Williams 2009] Conrad, P.; Shah, J.; and Williams, B. C. 2009. Flexible execution of plans with choice. In *ICAPS*.

[Despouys and Ingrand 1999] Despouys, O., and Ingrand, F. 1999. Propice-Plan: Toward a unified framework for planning and execution. In *ECP*.

[Doherty, Kvarnström, and Heintz 2009] Doherty, P.; Kvarnström, J.; and Heintz, F. 2009. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. *J. Autonomous Agents and Multi-Agent Syst.* 19(3):332–377.

[Effinger, Williams, and Hofmann 2010] Effinger, R.; Williams, B.; and Hofmann, A. 2010. Dynamic execution of temporally and spatially flexible reactive programs. In *AAAI Wksp. on Bridging the Gap between Task and Motion Planning*, 1–8.

[Feldman and Domshlak 2013] Feldman, Z., and Domshlak, C. 2013. Monte-carlo planning: Theoretically fast convergence meets practical efficiency. In *UAI*.

[Feldman and Domshlak 2014] Feldman, Z., and Domshlak, C. 2014. Monte-carlo tree search: To MC or to DP? In *ECAI*, 321–326.

[Ferrein and Lakemeyer 2008] Ferrein, A., and Lakemeyer, G. 2008. Logic-based robot control in highly dynamic domains. *Robotics and Autonomous Systems* 56(11):980–991.

[Firby 1987] Firby, R. J. 1987. An investigation into reactive planning in complex domains. In *AAAI*, 202–206. AAAI Press.

[Ghallab, Nau, and Traverso 2016] Ghallab, M.; Nau, D. S.; and Traverso, P. 2016. *Automated Planning and Acting*. Cambridge University Press.

[Hähnel, Burgard, and Lakemeyer 1998] Hähnel, D.; Burgard, W.; and Lakemeyer, G. 1998. GOLEX – bridging the gap between logic (GOLOG) and a real robot. In *KI*, 165–176. Springer.

[Ingham, Ragno, and Williams 2001] Ingham, M. D.; Ragno, R. J.; and Williams, B. C. 2001. A reactive model-based programming language for robotic space explorers. In *i-SAIRAS*.

[Ingrand and Ghallab 2017] Ingrand, F., and Ghallab, M. 2017. Deliberation for Autonomous Robots: A Survey. *Artificial Intelligence* 247:10–44.

[Ingrand et al. 1996] Ingrand, F.; Chatilla, R.; Alami, R.; and Robert, F. 1996. PRS: A high level supervision and control language for autonomous mobile robots. In *ICRA*, 43–49.

[James, Konidaris, and Rosman 2017] James, S.; Konidaris, G.; and Rosman, B. 2017. An analysis of monte carlo tree search. In *AAAI*, 3576–3582.

[Kocsis and Szepesvári 2006] Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *ECML*, volume 6, 282–293.

[Levine and Williams 2014] Levine, S. J., and Williams, B. C. 2014. Concurrent plan recognition and execution for human-robot teams. In *ICAPS*.

[Muscettola et al. 1998] Muscettola, N.; Nayak, P. P.; Pell, B.; and Williams, B. C. 1998. Remote Agent: To boldly go where no AI system has gone before. *Artificial Intelligence* 103:5–47.

[Myers 1999] Myers, K. L. 1999. CPEF: A continuous planning and execution framework. *AI Mag.* 20(4):63–69.

[Nau et al. 1999] Nau, D. S.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *IJCAI*, 968–973.

[Pollack and Horty 1999] Pollack, M. E., and Horty, J. F. 1999. There's more to life than making plans: Plan management in dynamic, multiagent environments. *AI Mag.* 20(4):1–14.

[Santana and Williams 2014] Santana, P. H. R. Q. A., and Williams, B. C. 2014. Chance-constrained consistency for probabilistic temporal plan networks. In *ICAPS*.

[Simmons and Apfelbaum 1998] Simmons, R., and Apfelbaum, D. 1998. A task description language for robot control. In *IROS*, 1931–1937.

[Simmons 1992] Simmons, R. 1992. Concurrent planning and execution for autonomous robots. *IEEE Control Systems* 12(1):46–50.

[Teichteil-Königsbuch, Infantes, and Kuter 2008] Teichteil-Königsbuch, F.; Infantes, G.; and Kuter, U. 2008. RFF: A robust, FF-based MDP planning algorithm for generating policies with low probability of failure. In *ICAPS*.

[Verma et al. 2005] Verma, V.; Estlin, T.; Jónsson, A. K.; Pasareanu, C.; Simmons, R.; and Tso, K. 2005. Plan execution interchange language (PLEXIL) for executable plans and command sequences. In *i-SAIRAS*.

[Wang et al. 1991] Wang, F. Y.; Kyriakopoulos, K. J.; Tsolkas, A.; and Saridis, G. N. 1991. A Petri-net coordination model for an intelligent mobile robot. *IEEE Trans. Syst., Man, and Cybernetics* 21(4):777–789.

[Williams and Abramson 2001] Williams, B. C., and Abramson, M. 2001. Executing reactive, model-based programs through graph-based temporal planning. In *IJCAI*.

[Yoon et al. 2008] Yoon, S. W.; Fern, A.; Givan, R.; and Kambhampati, S. 2008. Probabilistic planning via determinization in hindsight. In *AAAI*, 1010–1016.

[Yoon, Fern, and Givan 2007] Yoon, S. W.; Fern, A.; and Givan, R. 2007. Ff-replan: A baseline for probabilistic planning. In *ICAPS*, volume 7, 352–359.

# On Controllability of Temporal Networks: A Survey and Roadmap

**Jeremy D. Frank**
NASA Ames Research Center
jeremy.d.frank@nasa.gov

## 1 Introduction

Since its introduction by (Vidal and Ghallab 1996) and (Vidal and Fargier 1999), there has been considerable research in the area of *controllability* of temporal networks in the presence of *uncertainty*. Controllability asks: can events be scheduled to satisfy constraints in the presence of uncertain outcomes? The simplest problems assume no qualitative information is known about the timing of some events. More complex problems combine temporal constraints, uncertainty, and *preferences*. Uncertainty can be generalized so that algorithms must handle *probabilities* over when events occur. When it is impossible to ensure all constraints are satisfied, bounding below the probability of a constraint violation leads to new *risk-bounded* and *chance-constrained* problems; if the solution is still unsatisfactory, the constraints and risk bound can be *relaxed*.

Diverse though these problems are, there are gaps in existing research, as well as new problems to address. We systematically describe recent results on controllability of temporal networks with uncertainty in order to understand the current state of the art in controllability problems and algorithms, and rationalize prior definitions. We then examine results for these problems, and identify gaps in problems that have been studied. Finally, we provide recommendations for work in this area, both to study newly identified problems, and to revisit old problems with new approaches.

The scope of this paper is limited to generalizations of the Simple Temporal Network (STN) to include probability, bounds on failure, preferences, and costs on relaxation of constraints and the bounds on failure. While recent work also includes control of disjunctive temporal networks, conditional constraints, and partial observability, a more complete survey is left to future work.

## 2 Notation and Definitions

**Definition 1 (STN)** *(Dechter, Meiri, and Pearl 1991) Simple Temporal Networks (STNs) consist of timepoints $T$ with domain of $t_i \in T = \mathbb{R}$. and constraints $c(t_i, t_j)$ of the form $(t_j - t_i) \in [l_{t_i,t_j}, u_{t_i,t_j}]$.*

**Definition 2 (STNU)** *(Vidal and Ghallab 1996) (Vidal and Fargier 1999) Simple Temporal Networks with Uncertainty (STNUs) consist of* Activated *time-points $a_i$, i.e. those assigned by the agent, $A = \cup_i a_i$ and* received *time-points $r_i$, i.e. those assigned by the external world, $R = \cup_i r_i$. The set of timepoints $T = A \cup R$. The domain of $t_i \in T = \mathbb{R}$.* Free constraints *$c(t_i, t_j)$ have the form $(t_j - t_i) \in [l_{t_i,t_j}, u_{t_i,t_j}]$. Let $C = \cup_{t_i,t_j}\ c(t_i, t_j)$.* Contingent constraints *$g(a_i, r_j)$ have the form $(r_j - a_i) \in [l_{a_i,r_j}, u_{a_i,r_j}]$ where $a_i \in A$, $r_j \in R$; the semantics is that $\exists v \in [l_{a_i,r_j}, u_{a_i,r_j}] \mid r_j - a_i = v$ but $v$ is only observed during execution. Let $G = \cup_{a_i,r_j}$ $g(a_i, r_j)$. An STNU is a 4-tuple <A,R,C,G >.*

**Definition 3** *A schedule $s$ is an assignment to $a_i \in A$. Denote the value of $a_i$ in $s$ by $s(a_i)$. Denote by $S$ the set of all schedules.*

Ideally, a schedule works regardless of the uncertain outcomes in an STNU. A less stringent requirement is to generate a *strategy* that reacts to observed events to ensure constraints are satisfied. These ideas are formalized in the definitions of *controllability* of STNUs.

**Definition 4 (Controllability of STNU)** *(Vidal and Fargier 1999) Let $P$ be an STNU. Let $V = \times_{g_{a_i,r_j}}$ $[l_{a_i,r_j}, u_{a_i,r_j}]$ (the cross product of all possible uncertain outcomes of all contingent constraints). $P$ is* Strongly Controllable *(SC) if there is a schedule $s$ such that $\forall v \in V$, $s$ satisfies all constraints $c(t_i, t_j)$. Denote the time a received timepoint $r_i$ occurs and is observed by $v(r_i)$; denote the time a controllable event $a_i$ is executed by $e(a_i)$. $P$ is* Dynamically Controllable *(DC) if there is an* execution strategy *$\pi_{dc}$ satisfying all constraints $c(t_i, t_j)$, such that $e(a_i)$ derived from $\pi_{dc}$ may depend only on previously observed uncontrollable event occurrences $v(r_j) \leq e(a_i)$.*

Controllability in the presence of *preferences* typically must achieve the best possible schedule, with an option to 'ignore' outcomes that lead to preferences below a threshold, as formalized below:

**Definition 5 (STPPU)** *(Rossi, Venable, and Yorke-Smith 2006) Let $f_s$ be a preference function on schedules. A Simple Temporal Problems with Preferences and Uncertainty (STPPU) is a 5-tuple <A,R,C,G,$f_s$ >.*

**Definition 6** *Let $P$ be an STPPU. Let $V = \times_{g_{a_i,r_j}}$ $[l_{a_i,r_j}, u_{a_i,r_j}]$ (the cross product of all possible uncertain outcomes of all contingent constraints). $f_s(s, v)$ is the value of a schedule $s$ combined with a set of outcomes $v \in V$. Let $f_s(opt, v) = \max_s f_s(s, v)$ be the value of the best schedule given outcome $v$.*

**Definition 7 (Controllability of STPPUs)** *(Rossi, Venable, and Yorke-Smith 2006) Let $P$ be an STPPU. $P$ is* Optimally Strongly Controllable *(OSC) if there is a schedule $s$ such that $\forall v \in V$, $s$ satisfies all constraints $c(a_i, r_j)$ and $s$ is optimal for each $v \in V$ (that is, $\forall v \in V, (\forall s' \in S(f_s(s', v) \leq f_s(s, v)))$.*

*$P$ is* $\alpha$-Strongly Controllable *($\alpha$-SC) if, $\forall v \in V$, $s$ satisfies all constraints $c(a_i, r_j)$ and $f_s(opt, v) \leq \alpha \Rightarrow f_s(s, v) \leq f_s(opt, v)$.*

*$P$ is* Optimally Dynamically Controllable *(ODC) if there is an* execution strategy $\pi_{odc}$ *satisfying all constraints $c(t_i, t_j)$ such that, for $v_1, v_2 \in V$, 1) $s(a_i)$ derived from $\pi_{odc}$ may differ depending only on previously observed uncontrollable event occurrences in $v_1, v_2$, 2) $s_1$ is a schedule consistent with $v_1$ and $s_2$ is a schedule consistent with $v_2$ and 3) $f_s(s_1, v_1)$ and $f_s(s_2, v_2)$ are optimal.*

*$P$ is* $\alpha$-Dynamically Controllable *($\alpha$-DC) if if there is an* execution strategy $\pi_{\alpha dc}$ *satisfying all constraints $c(t_i, t_j)$ such that, for $v_1, v_2 \in V$, 1) $s(a_i)$ derived from $\pi_{\alpha dc}$ may differ depending only on previously observed uncontrollable event occurrences in $v_1, v_2$, 2) $s_1$ is a schedule consistent with $v_1$ and $s_2$ is a schedule consistent with $v_2$ and 3) $f_s(s_1, v_1) \leq \alpha$ and $f_s(s_2, v_2) \leq \alpha$*

PSTNs generalize STNUs by adding probability of duration. Typical approaches transform a PSTN into an STNU and then evaluate controllability. *Risk* describes the probability that, given a schedule or strategy, an outcome $v \in V$ violates some constraint. To compute risk for an STNU, we must measure how much probability mass is not covered after 'squeezing' to transform it into a contingent link, i.e. transforming $d(a_i, r_j)$ to $g(a_i, r_j)$. We formalize the 'squeeze' operation below.

**Definition 8 (PSTN)** *(Tsamardinos 2002) Let a* probabilistic duration constraint $d(a_i, r_j) : \Omega_{a_i, r_j} \rightarrow \mathbb{R}^+$ *be a random variable describing the probability of the difference $r_j - a_i$, $P(v(r_j) - s(a_i) = \omega)$, where $a_i \in A$, $r_j \in R$. Let $D = \cup_{a_i, r_j} d(a_i, r_j)$. (Duration constraints $d(r_i, r_j)$ are also permitted.) A Probabilistic Simple Temporal Networks (PSTN) is a 4-tuple <A,R,C,D>.*

**Definition 9** *Let $\rho_d$: $D \Rightarrow G$ transform a duration constraint into a contingent link by choosing a compact subset $[l_{a_i, r_j}, u_{a_i, r_j}] \subset \Omega_{r_i}$. Let $\rho_D = \{\rho_d\}$. Let $P$ be a PSTN. Then $\rho_D(P) = P'$ where $P'$ is the STNU derived from $P$.*

**Definition 10** *Let $P$ be a PSTN. Let $P' = \rho_D(P)$ be an STNU derived from $P$. Let $\rho_d(d(a_i, r_j)) = g(a_i, r_j)$. Let $[l_{a_i, r_j}, u_{a_i, r_j}]$ be the contingent constraint interval defined by $g(a_i, r_j)$. Let $\Phi_g = \omega \in \Omega_{r_i} | \, \omega \leq l_{a_i, r_j}$. Let $\Theta_g = \omega \in \Omega_{r_i} | \, \omega \geq u_{a_i, r_j}$. The* risk *of $d(a_i, r_j)$ relative to $\rho_d$, denoted $\delta(\rho_d, d(a_i, r_j))$, is $\int_{\omega \in \Phi_g \cup \Theta_g} P(\omega)$. The symmetric case of $d(r_i, a_j)$ is similar. The* risk *of $P$ relative to $\rho_D$, denoted $\delta(P, \rho_D)$, is $1 - \left(\prod_{d \in D}(1 - \delta(\rho_d, d(t_i, t_j)))\right)$.*

**Definition 11** *$P$ is SC with risk $\Delta$ if $\exists \, P' = \rho_D(P)$, $P'$ is SC, and $\delta(P, \rho_D) = \Delta$. $P$ is DC with risk $\Delta$ if $\exists \, P' = \rho_D(P)$, $P'$ is DC, and $\delta(P, \rho_D) = \Delta$.*

PSTN problems can now be characterized as searching over $\rho_D$ in order to minimize $\delta(P, \rho_D)$, or satisfying some risk bound $\Delta$. While (Wang and Williams 2015) and (Lund et al. 2017) (implicitly) address this problem, it has not been crisply distinguished from finding the most preferred schedule while bounding the risk. For this reason, we introduce a new definition:

**Definition 12 (BPSTN)** *A (Risk-) Bounded PSTNs (BPSTN) is a PSTN and a risk bound $\Delta$, thus, a 5-tuple <A,R,C,D,$\Delta$ >.*

**Definition 13 (CCPSTN)** *(Fang, Yu, and Williams 2014) A Chance-Constrained PSTN (CCSTN) is a PSTN, a function $f_s : S \rightarrow \mathbb{R}$, and a risk bound $\Delta \in [0, 1]$, thus, a tuple <A,R,C,D,$\Delta$,$f_s$ >.*

The RCCPSTN of (Yu, Wang, and Williams 2015) allows relaxation of the free constraints and the risk bound. These notions are defined somewhat informally; a slightly more formal definition is provided here.

**Definition 14** *A* relaxation *of a constraint $c(t_i, t_j)$ with bound $[l_{t_i, t_j}, u_{t_i, t_j}]$ is defined as a new bound $[l'_{t_i, t_j}, u'_{t_i, t_j}]$ such that $l'_{t_i, t_j} \leq l_{t_i, t_j}$ and $u'_{t_i, t_j} > u_{t_i, t_j}$ or $l'_{t_i, t_j} < l_{t_i, t_j}$ and $u'_{t_i, t_j} \geq u_{t_i, t_j}$; the set of relaxations is denoted $R(c)$. A* relaxation *of $\Delta$ is $\Delta' > \Delta$. Denote a PSTN $P^r$ that is a relaxation of $P$ by $P \subset P^r$.*

**Definition 15 (RCCPSTN)** *(Yu, Wang, and Williams 2015) A Relaxable CCPSTNs (RCCPSTN) is a PSTN, a risk bound $\Delta$, a set of functions $f_c : c' \in R(c) \rightarrow \mathbb{R}^+$, and a function $f_\Delta : [0, 1] \rightarrow \mathbb{R}^+$, thus, a tuple <A,R,C,D,$\Delta$,$\{f_c\}$, $f_\Delta$ >.*

Notes: We define CCPSTNs to include $f(s)$ (schedule optimization) because (Fang, Yu, and Williams 2014) was published first; we introduce the name BPSTN to distinguish CCPSTNs from the problem of only minimizing or bounding risk. (Wang and Williams 2015) generalize PSTNs by permitting multiple risk bounds over subplans; we have not included this PSTN variant in our definitions. The original definitions of RCCPSTNs and PSTNs allows bounding the risk on a subset of the constraints ((Fang, Yu, and Williams 2014) p. 4); other constraints may have arbitrary risk. We have not made this minor nuance explicit in our definitions. The definition of RCCPSTNs in (Yu, Wang, and Williams 2015) restricts the set of constraints that can be relaxed; our definition ensures there is a relaxation function on all the links and $\Delta$. Finally, we have omitted the PSTNUs of (Santana et al. 2016) as they are a minor elaboration of PSTNs.

Table 1 surveys previous work in controllability. Figure 1 shows the relationship between the different controllable problems described in the definitions. We use the definitions OSC, $\alpha$-SC, ODC and $\alpha$-DC notation from (Rossi, Venable, and Yorke-Smith 2006) for all problems, even CCPSTNs, in deference to their early work.

## 3 Unaddressed Problems, Theoretical Analysis, and Empirical Studies

In this section we discuss unaddressed problems, theoretical analysis, and empirical studies that should be undertaken to advance work in controllability.

| Input | Ctrl | Risk | Opt | Notes |
|---|---|---|---|---|
| STNU | SC | N/A | N/A | Polynomial time ($O(n^3)$, (Morris 2014)). |
| STNU | DC | N/A | N/A | Polynomial time ($O(n^3)$, (Morris 2014)). |
| STPPU | OSC | N/A | $f_s$ | Tractable in limited cases (semi-convex preferences on c-semirings) (Rossi, Venable, and Yorke-Smith 2006) |
| STPPU | ODC | N/A | $f_s$ | Tractable in limited cases (semi-convex preferences on c-semirings) (Rossi, Venable, and Yorke-Smith 2006) |
| STPPU | $\alpha$-SC | N/A | $f_s$ | Tractable in limited cases (semi-convex preferences on c-semirings) (Rossi, Venable, and Yorke-Smith 2006) |
| STPPU | $\alpha$-DC | N/A | $f_s$ | Tractable in limited cases (semi-convex preferences on c-semirings) (Rossi, Venable, and Yorke-Smith 2006) |
| PSTN | SC | $\min_s \delta(P, \rho_D)$ | N/A | General optimization. Local Optimal only. (Tsamardinos 2002) |
| PSTN | SC | $\min_s \delta(P, \rho_D)$ | N/A | Polynomial time (complexity depends on size of LP). Sound but incomplete: risk is bounded above using piecewise constant bounds on probabilities, complete for uniform distributions. Empirical results on Gaussian distributions. (Santana et al. 2016) |
| PSTN | DC | $\min_{\rho_D} \delta(P, \rho_D)$ | N/A | General optimization. Informal algorithm using SC algorithm described, no results. (Tsamardinos 2002) |
| BPSTN | SC | $\exists \rho_D \mid \delta(P, \rho_D) \leq \Delta$ | N/A | General optimization. Conflict-driven search over risk. Nonlinear solver allocates risk, analysis of infeasible STNUs generates conflicts that drive repeated nonlinear search; number of nonlinear solve operations polynomially bounded. Exact probability of success computed, requires conditionally independent probabilities. (Wang and Williams 2015) |
| BPSTN | SC | $\exists s \mid \delta(P, \rho_D) \leq \Delta$ | N/A | Pseudo-polynomial time; binary search over relaxations of probabilistic ranges, complexity depends on size of LP. Sound but incomplete, uses LP approximation of probabilities (nonbounding constant approximation of tails of distributions) Risk roughly equally allocated over each tail of each probability. Exact probability of success computed, requires conditionally independent probabilities. Empirical results on Gaussian distributions with varying $\sigma$. (Lund et al. 2017) |
| BPSTN | DC | $\exists \rho_D \mid \delta(P, \rho_D) \leq \Delta$ | N/A | Not considered. |
| CCPSTN | OSC | $\delta(P, \rho_D) \leq \Delta$ | $f_s$ | General optimization. Optimal, risk is bounded above by $\sum_{d \in D} \delta(\rho_d, d(t_i, t_j))$, arbitrary (including joint) probability distributions. Empirical results on Gaussian distributions; objective is makespan. (Fang, Yu, and Williams 2014) |
| CCPSTN | OSC | $\delta(P, \rho_D) \leq \Delta$ | $f_s$ | Polynomial time (complexity depends on size of LP formed). Sound but incomplete in general: risk is bounded above using LP approximation of probabilities (piecewise constant), complete for uniform distributions. Empirical results on Gaussian distributions; objective is schedule makespan. (Santana et al. 2016) |
| CCPSTN | $\alpha$-SC, | $\delta(P, \rho_D) \leq \Delta$ | $f_s$ | Not considered |
| CCPSTN | ODC, | $\delta(P, \rho_D) \leq \Delta$ | $f_s$ | Not considered |
| CCPSTN | $\alpha$-DC, | $\delta(P, \rho_D) \leq \Delta$ | $f_s$ | Not considered |
| RCCPSTN | SC | $\delta(P, \rho_D) \leq \Delta$ | $f_c$ | Not considered |
| RCCPSTN | DC | $\delta(P, \rho_D) \leq \Delta$ | $f_c$ | General optimization. Branch and bound on cost of fixing conflicts. Risk is bounded above, arbitrary functional form of probability distributions. Empirical results on Gaussian and Uniform distributions. (Yu, Wang, and Williams 2015) |

Table 1: Previous work in controllability.

### 3.1 Unaddressed Problems

Consider the problem of finding $P' = \rho_D(P)$ such that $P'$ is DC for a BPSTN $P$. While unaddressed, the algorithms described in (Santana et al. 2016), (Wang and Williams 2015) or (Lund et al. 2017) can be adapted to address this problem with relative ease. This is because all search over $\rho_D$, in different ways, then solve a sub-problem to find the best SC schedule; the sub-problem can instead be a DC check. While seemingly straightforward, doing so would fill a gap in controllability research.

Of more interest, there are several unaddressed problems in finding an *optimal* DC policy for PSTNs and their descendants. Initially, one could apply the tractability results for STPPUs with semirings (Rossi, Venable, and Yorke-Smith 2006) to obtain new algorithms and results. However, more general preference functions should be considered as the foundation for optimal DC problems, such as the additive convex preferences of (Morris et al. 2004).

### 3.2 Controlling for Expected Value

Suppose that the smallest risk for which a controllable STNU can be extracted from a PSTN is 'too high'. This leads to an undesirable states of affairs; either there is no solution to the PSTN, the RCCPSTN must be used to relax some or all of the constraints, or at execution time, it is very likely that a bad outcome $v \in V$ will break the strategy. An alternative solution to this problem is to try to satisfy as many constraints as possible at execution time.

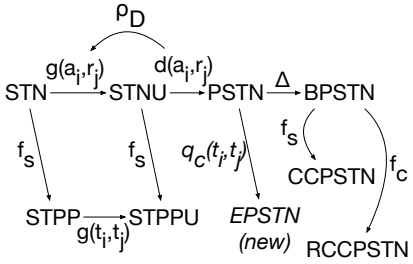



Figure 1: PSTN 'family tree', including the new EPSTN.

If some constraints are more important than others, then a natural optimization criteria for the strategy is to maximize the *expected value* of satisfied constraints. This new problem blends several notions explored in the controllability literature to date. Accepting risk implies accepting outcomes that violate some constraints. Applying preferences to satisfied constraints suggests control of expected schedule quality based on past information and the probability and cost of future constraint violations.

Denote by $q_c(t_i, t_j) \in \mathbb{R}^+$ the value of satisfying constraint $c(t_i, t_j)$. Given a PSTN $P$, an STNU $P' = \rho_D(P)$, satisfying some risk bound $\Delta$, we can compute the *expected value* of the schedule; with some more work, we should be able to do the same for a strategy. This simple Expected Value PSTN (EPSTN) can then be generalized with more sophisticated preferences similar to those used in (Rossi, Venable, and Yorke-Smith 2006) and (Morris et al. 2004).

### 3.3 Empirical Analysis

(Santana et al. 2016) and (Lund et al. 2017) are similar algorithms for solving controllability problems on PSTNs, in that they both search over $\rho_D$ to find STNUs, then employ LP formulations to find an SC plan; a non-bounding approximation of risk is used in (Lund et al. 2017), a piecewise constant bound in (Santana et al. 2016). Other LP formulations are possible, e.g. a piecewise linear approximation of the probability distribution over $d(t_i, t_j)$ guaranteed to be convex, using LP formulations in (Frank et al. 2006) and (Morris et al. 2004), that directly captures the risk of a schedule. All approaches trade low computational complexity for a loose bound on risk (compared to more powerful nonlinear optimizers). Some approaches may employ larger LPs but bound risk more tightly, and perhaps eliminate outer-loop search. A head-to-head comparison of these three approaches should be done, focusing on tradeoffs of the true risk achieved vs computation time.

In STNUs, there is no theoretical advantage to precomputing the DC policy vs re-solving after new information is achieved, because there is a response to all possible uncertain outcomes. When minimizing or bounding the risk of a PSTN, recomputation of policies can recover from bad outcomes (those violating constraints in the STNU), or reduce or redistribute the risk when new information is available. DREA (Lund et al. 2017) recomputes the policy when new information is achieved; it is therefore unsurprising that it outperforms algorithms like SREA and DC. The recomputation strategy for DREA should be extended to other PSTN approaches, e.g. PARIS (Santana et al. 2016); even simple earliest first with recomputation may be a viable strategy.

Distributions used in prior empirical studies have been limited to Gaussians and Uniform. Skewed distributions, heavy-tailed distributions, and bimodal distributions should also be studied. Joint probability distributions should be studied, e.g. joint probability over event start time and duration. Such distributions should cause fewer problems for a recomputation strategy like DREA, as discussed above, but may lead to poor true risk for static strategies. Study of such distributions may also lead to a principled algorithm to produce a DC policy for joint distributions. Finally only a few benchmarks exist (ride sharing (Fang, Yu, and Williams 2014), UAV (Yu, Wang, and Williams 2015), simple multi-robot domains (Lund et al. 2017)). More benchmarks, with different structural properties, should be created.

### 3.4 Other Theoretical Considerations

STPPs with convex additive preferences as described in (Morris et al. 2004) can be modeled as LPs using piecewise linear functions. Additive preferences are more satisfactory than the 'max' semirings used to develop previous STPPU theory (Rossi, Venable, and Yorke-Smith 2006). The theory of STPPUs should be extended using such preferences; this may lead to tractable SC and DC results for STPPUs with more intuitive and useful preference functions.

The usual means of measuring the *flexibility* of a schedule $s$ that addresses the constraints in a scheduling problem is to evaluate the solution space of the STN derived from the activity ordering decisions; new work in this area appears in the main conference (Huang et al. 2018). However, treating the uncontrollables as controllables in order to measure flexibility is inappropriate. Fundamentally, flexibility's purpose is to allow a policy that minimizes *risk*. If the original scheduling problem with uncertainty can produce a PSTN or one of its variants, instead of an STN, we can now pose and solve problems such as BPSTN to find the minimum risk STNU, formally defined by $\delta(P, \rho_D)$, for a given PSTN. We can then look for the schedule whose PSTN achieves the minimum risk STNU in an 'outer loop' search.

It may be possible to *transform* an RCCPSTN into a CCPSTN. One transformation (linear decreasing preference functions for values below or above the bound) leads to piecewise constant, linear $f_s$; such functions can be posed and solved in CCPSTNs. Other transformations may be more complex. If the RCCPSTN is preferred, the reverse transformation is also worth exploring.

Finally, suppose a BPSTN or other variant of a PSTN has multiple STNUs of equal risk. Other properties of the STNUs, derived from the approximation, could be identified to select the best solution. For instance, is it better to try evenly distributing risk among uncertain durations as a secondary criteria?

## References


Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence* 49:61–94.

Fang, C.; Yu, P.; and Williams, B. 2014. Chance-constrained probabilistic simple temporal problems. In *Proceedings of the National Conference on Artificial Intelligence*, 2264 – 2270.

Frank, J.; Crawford, J.; Khatib, L.; and Brafman, R. 2006. Tractable optimal competitive scheduling. In *Proceedings of the* $16^{th}$ *International Conference on Automated Planning and Scheduling*, 73 – 82.

Huang, A.; Lloyd, L.; Omar, M.; and Boerkoel, J. C. 2018. New perspectives on flexibility in simple temporal planning. In *Proceedings of the* $28^{th}$ *International Conference on Automated Planning and Scheduling*.

Lund, K.; Dietrich, S.; Chow, S.; and Boerkoel, J. 2017. Robust execution of temporal plans. In *Proceedings of the National Conference on Artificial Intelligence*, 3597 – 3604.

Morris, P.; Morris, R.; Khatib, L.; Ramakrishnan, S.; and Bachmann, A. 2004. Strategies for global optimization of temporal preferences. In *Proceedings of the* $10^{th}$ *International Conference on the Principles and Practices of Constraint Programming*, 408 –422.

Morris, P. 2014. Dynamic controllability and dispatchability relationships. In *Proceedings of the IInternational Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 464 – 479.

Muscettola, N.; Morris, P.; and Vidal, T. 2001. Dynamic control of plans with temporal uncertainty. In *Proceedings of the* $17^{th}$ *International Joint Conference on Artificial Intelligence*.

Rossi, F.; Venable, K. B.; and Yorke-Smith, N. 2006. Uncertainty in soft temporal constraint problems: A general framework and controllability algorithms for the fuzzy case. *Journal of Artificial Intelligence Research* 27:617–674.

Santana, P.; Vaquero, T.; Toledo, C.; Wang, A.; and Williams, B. 2016. Paris: A polynomial-time, risk-sensitive scheduling algorithm for probabilistic simple temporal networks with uncertainty. In *Proceedings of the National Conference on Artificial Intelligence*, 267 – 275.

Tsamardinos, I. 2002. A probabilistic approach to robust execution of temporal plans with uncertainty. In *Methods and Applications of Artificial Intelligence*, 97 – 108.

Vidal, T., and Fargier, H. 1999. Handling contingency in temporal constraint networks: from consistency to controllabilities. *Journal of Experimental and Theoretical Artificial Intelligence* 11(1):23 – 45.

Vidal, T., and Ghallab, M. 1996. Dealing with uncertain durations in temporal constraint networks dedicated to planning. In *Proceedings of the* $12^{th}$ *European Conference on Artificial Intelligence*, 48 – 54.

Wang, A. J., and Williams, B. 2015. Chance-constrained scheduling via conflict-directed risk allocation. In *Proceedings of the National Conference on Artificial Intelligence*, 3620 – 3627.

Yu, P.; Wang, C.; and Williams, B. 2015. Resolving over-constrained probabilistic temporal problems through chance constraint relaxation. In *Proceedings of the National Conference on Artificial Intelligence*, 3425 – 3431.

# Task Monitoring and Rescheduling for Opportunity and Failure Management

**José Carlos González**
josgonza@inf.uc3m.es
Universidad Carlos III de Madrid (UC3M)
Leganés, Madrid, Spain

**Manuela Veloso**
mmv@cs.cmu.edu
Carnegie Mellon University (CMU)
Pittsburgh, Pennsylvania, USA

**Fernando Fernández** and **Ángel García-Olaya**
{ffernand, agolaya}@inf.uc3m.es
Universidad Carlos III de Madrid (UC3M)
Leganés, Madrid, Spain

### Abstract

The CoBot robots, as other service robots, autonomously navigate in building environments performing different types of tasks that include item transportation and person guiding between locations. The CoBots can execute their planned routes, localize in the environment, avoid obstacles, and ask for help to humans to overcome their actuation limitations. However, they were not able to handle high-level unexpected events during execution, such as interruptions with new task requests that may need a careful analysis of rescheduling trade-offs. Unexpected events can be failures if their influence is on the pending tasks or opportunities if it is on the robot expectations. This work presents a new task-execution, monitoring, and rescheduling architecture, which includes a representation of new task features to be monitored to detect failures and opportunities, as well as a task scheduler to evaluate time and task features constraints. We demonstrate the new features in a task that needs to deliver hot coffee at some time, noting that the coffee gets cold with interruption delays.

## Introduction

The development of autonomous service robots has been an important research topic in recent times. One notable example is the CoBot robots. CoBots are autonomous agents that exploit their presence to interact with people (Veloso et al. 2015). Their tasks involve physical movement from one place to another and different kinds of physical interactions with the real world, as to deliver a message or an object, to go to another place, to escort someone to an office, etc. The nature of these tasks is sequential and they cannot overlap.

Currently, a CoBot is basically a mobile robotic base with a computer, several sensors and a surface or basket to store objects. Any task that requires physical manipulation, like opening doors, pushing elevator buttons or getting objects, requires human help. CoBots are designed to ask for help to nearby people if they do not have the capability to perform a certain action of a task (Rosenthal and Veloso 2012). Tasks are created by the users using a web interface (Coltin, Veloso, and Ventura 2011) or just using the embedded touchscreen. Users can assign several time constraints to tasks. The robot has a task scheduler module in charge of finding a plan to perform all the operations. If the new task cannot be scheduled, the user is asked to change the parameters.

The scheduler can be considered as a high-level planner in a multi-level architecture like NAOTherapist (González, Pulido, and Fernández 2017). This hierarchical view of robotic architectures has been long discussed in the literature (Ghallab, Nau, and Traverso 2014). NAOTherapist uses a triple-layer planning mechanism where a high-level planner generates a plan to determine the tasks, or high level goals, to perform in a session (exercises to do). Traditionally, this high level works in an offline phase, without any rescheduling abilities. Then, a medium level controls and monitors the actions for each planned task to face unexpected events during their execution (movements of each exercise), sometimes even interrupting the current action. The low level acts like a path planner for each action, working directly with the robot platform.

CoBots can also replan their behavior in these medium and low levels of planning, avoiding obstacles, canceling a task if the target person is not in the place, etc. However, there are high-level events out of the control of these robots. One clear example arises when the robot has to deliver a spoken message to someone. It will try to find the recipient of the message in her office, but if, by chance, the person appears near the robot in its way, it will just continue driving to the office wasting the *opportunity* to deliver the message at that moment. The situation, in fact, will probably end up with a task fail since the person will not be at the office when the robot arrives. A *failure* also happens if the robot has to deliver an object, but loses it in the way. Therefore, CoBot was not able to tag these high-level events as opportunities or failures to avoid wasting time in unnecessary operations.

Being able to detect and act accordingly to some of these events is important to increase the autonomy of a robotic platform like CoBot. The automated planning field (Ghallab, Nau, and Traverso 2004) has approaches dealing with opportunities and failures. There are works about fast recovering of failures (Guzmán et al. 2015; Alcázar et al. 2010), which interleave planning and execution although they are more focused in the middle and low levels of planning. Other approaches use goal management techniques to take advantage

of some detected opportunities (Schermerhorn et al. 2009) by triggering soft goals at certain moments. Also, temporal planning has been used in opportunistic planning (Cashmore et al. 2017) approaches. However, the task scheduling problem addressed in this manuscript is heavily based on numbers and currently automated planning does not have good heuristics for numerical fluents.

Planning a schedule is a problem of mathematical optimization. There can be many feasible schedules for the same task pool with very different quality. The scheduler must try to find not only a suitable schedule, but also the optimal one in a limited amount of time. This time depends on the response time requirements of each application. All numeric variables used in CoBots can be restricted to integers without affecting the quality of the predictions, so Mixed Integer Programming (MIP) (Chen, Batson, and Dang 2010) is a useful technique for this work. This is the same technique used in the original CoBot scheduler (Coltin, Veloso, and Ventura 2011), but this work greatly improves the capabilities of that MIP model.

The main paper contribution is a novel architecture for a task scheduler that is able to detect and manage some opportunities and failures. It can also manage complex time constraints like delivering a coffee before it gets cold (cooldown time). Furthermore, this component can be reused in other multilayer architectures to control high-level events by defining only the interfaces and the available tasks types. With this scheduler, the high-level module gets also the ability to replan according to new high-level events, bringing it to an online phase allowing it changing its tasks and schedules dynamically.

## High-level Task Scheduler Architecture

In CoBots, the scheduler is in charge of generating a valid schedule for the pool of remaining tasks and also managing their monitoring and execution to reschedule according to opportunities and failures. Figure 1 shows the different modules of the architecture of the developed component. Basically, users send tasks through an interface to the scheduler. Monitoring checks if a new task is valid by generating a MIP problem with some static data gathered from a shared knowledge base (for example distance among all involved locations). Then it sends this problem to an external solver which returns a valid schedule, if possible, in a certain amount of time. The tasks that are not executed yet are stored in the task pool. Afterward, the next task is sent to Execution when needed. Execution is just an interface for lower levels of planning of the robot. The robot constantly informs about the current state of the world. Monitoring gets these states and reasons if an opportunity or a failure has appeared, taking then the appropriate actions for each case. The rest of the CoBot works in a lower level, so the particular way it has to achieve each task is transparent for the scheduler.

High-level events can affect not only the movement or even the current task, but also future tasks in the schedule. These can be classified as opportunities and failures that must be detected by the robot in order to interrupt the execution if needed. They may not be easy to detect since they do not affect the execution of current medium or low-level

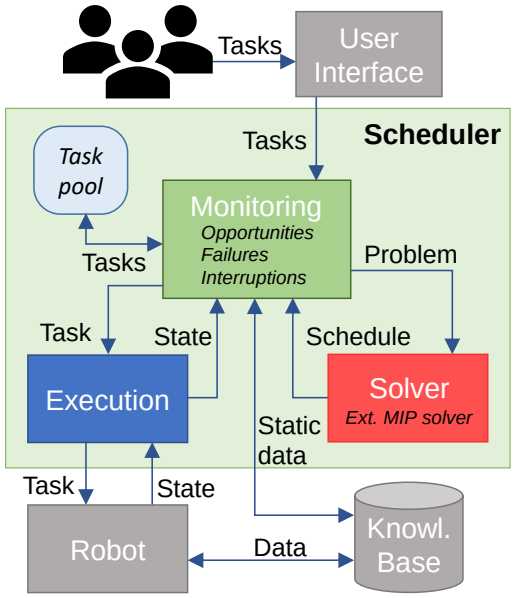



Figure 1: Architecture of the high-level task scheduler component.

actions. The need for the different modeled task parameters is motivated with a guiding example: delivering a hot coffee to someone.

## Modeling of the Tasks

Tasks are modeled as a sequence of subtasks with different parameters. A subtask is a part of the task that can be interleaved with subtasks from other tasks. In CoBots, the size of the sequence is determined by the times that the robot must change the location sequentially to accomplish the task. For each task and subtask, several parameters must be specified, as described in Table 1, which shows the decomposition of the DeliverDrink task. The task decomposition and the parameters (task type, type owner, etc.) are specified by the designer of the robotic use case, so the architecture can reason with them. The meaning of each parameter is specified in the following paragraphs. The parameter set is enough for the example followed in this paper, but should be updated for different tasks and/or use-cases.

The developed graphic user interface (GUI) eases the insertion of high-level tasks like delivering a coffee. It procedurally decomposes each possible task type into the subtasks, according to the task description defined by the designer. Breaking them into parts permits to interleave subtasks of different tasks to improve the global performance of the executed schedule. Although delivering a coffee is used in this manuscript as an example, this mechanism can be used to model any other task that has a part that must be finished before a time threshold, and could be extended to more complex decompositions with three or more subtasks.

In the example, the user sets the time window in which the task can be executed, while the GUI gathers the information from a knowledge base to fill optional parameters,

| | | Task | Subtask-1 | Subtask-2 |
|---|---|---|---|---|
| *User* | Task type | DeliverDrink | MakeHotDrink | DeliverObject |
| | Task owner | Alice | *Alice* | *Alice* |
| | Location start | - | CoffeMaker | CoffeMaker |
| | Location end | - | CoffeMaker | AliceOffice |
| | Time start min | 0 | *0* | *0* |
| | Time end max | 15 | *15* | *15* |
| | Person target | Alice | - | Alice |
| | Object | HotCoffee | HotCoffee | HotCoffee |
| | Priority | - | 10 | 10 |
| *Internal* | Time operation | - | 5 | 2 |
| | Time cooldown | - | - | 6 |
| | Task depending | - | - | Subtask-1 |
| | Opportunities | VIP | HotCoffee, *VIP* | Person target, *VIP* |
| | Failures | TO, BP | *TO, BP* | HotCoffee, *TO, BP* |

Table 1: Task model instantiated in the decomposition of a DeliverDrink task type. Inherited parameters are in italic.

like location start with Alice's usual office and the nearest coffee machine. The GUI also fills internal parameters like an estimation of the operation time and the cooldown time. A subtask can only be executed after the subtask indicated in *Task depending* is finished. The task types used in this work are: go to location, deliver message, telepresence, attract attention, deliver object, deliver drink, escort someone, wait for emergency services and recharge battery.

## Failures

A failure is a situation that impedes the success of the current task. It may require some actions to correct the problem or to abort the task if it is not feasible now. A *generic* failure can appear when a subtask cannot be completed because of blocked paths (BP) or a timeout (TO) due too many accumulated execution delays. Sometimes tasks have some *specific* failures. In the coffee example, the receipt could be indeed in his office, but the coffee could be stolen by someone from the basket. The internal parameter *Failures* indicates which other parameters must be invariant along the execution of the subtask. In Subtask-2 the robot must have *HotCoffee* in the basket at all times, as defined in Table 1.

## Opportunities

In the DeliverDrink example, a *specific* opportunity appears when the robot is going to a room with the hot coffee in the basket and finds the target person in the corridor. It can try to deliver the coffee in that moment. This allows to finish the current task earlier and probably avoiding a future failure. The *Opportunity* internal parameter indicates which other parameters must be monitored along the execution of the subtask to detect potential specific opportunities, as shown in Table 1. A *generic* opportunity (valid for all tasks) can also improve the whole schedule. For instance, CoBot could identify a very important person (VIP) during the execution of the schedule. The robot could approach the VIP to greet him or just ask him whether it can help somehow. Spending time on this opportunity will delay the rest of the scheduled tasks, but the gain obtained by executing this important task could worth it if it has enough *Priority*.

## Interrupting Tasks

Generic opportunities and failures must be defined in Monitoring for each application domain. For CoBots, a high priority VIP task is added in the task pool when the sensors identify a VIP in its surroundings. The internal predicates *Failures* and *Opportunities* control the specific high-level events to be detected for each subtask during all their execution.

Decomposing tasks in subtasks is very useful when interrupting high-level tasks. In the coffee example, if the robot already has the hot coffee in its basket and a VIP appears, Monitoring detects it as an opportunity. If the robot interrupts the coffee delivering to talk with the VIP, the coffee could be delivered cold. This is controlled with the *Cooldown time* internal parameter, which indicates the maximum time allowed after *Task depending* was finished.

In the DeliverDrink example, a change in the task pool (VIP) can force the robot to reason about the convenience of scheduling or not a certain task after making the coffee (CoffeeA) and before its delivery (CoffeeB). In this work there can be:

- **Redoing CoffeeA**. "I prefer to remake the coffee. The VIP can't wait, and I can deliver the coffee later"
- **VIP between CoffeeA and CoffeeB**. "I can spend some time talking with the VIP and then deliver the coffee on time"
- **VIP after CoffeeB**. "I can't redo the coffee later, but the VIP will be there for some time"
- **Cancel DeliverDrink**. "The VIP can't wait, and I can't redo the coffee later either. I prefer to cancel the coffee and talk with VIP"
- **Cancel VIP**. "The VIP can't wait, and I can't redo the coffee later either. I prefer to continue with my previous task and omit the VIP"

The result will depend on the gain that it will get after completing the schedule. This gain is modeled in the scheduler to allow the system to reason over these concepts.

## Numerical Predictions for Tasks

The problems addressed in this work are deeply based on numbers. The constraints that involve the ordering of tasks depend mainly on the starting and ending time of them. The times indicated in a schedule are, in the best case, just a good prediction of their values. In particular, distance measures can also be expressed in time units because the average speed of the robot along each path is known, as well as the average time to perform certain operations like asking someone, preparing a drink, etc.

In fact, planning a schedule is a problem of mathematical optimization. There can be many feasible schedules for the same task pool with very different quality. The scheduler must try to find not only a suitable schedule, but also the optimal one in a limited amount of time. This time depends on the response time requirements of each application.

All numeric variables used in the scheduler can be restricted to integers without affecting the quality of the predictions, so Mixed Integer Programming (MIP) (Chen, Batson, and Dang 2010) is a useful technique for this work. It

allows finding solutions to numerical optimization problems thanks to mechanisms like simplex. This is the same technique that the original scheduler used (Blind Reference 3). A MIP solver receives the input data and uses a model which describes all needed mathematical constraints to find a valid schedule. The quality of the solution is evaluated using an objective function, so after finding a valid one the scheduler can continue the searching until it can demonstrate that the last was optimal.

In the MIP aspect, this work contributes with a new way to solve a kind of temporal problem (cooling down time) with it, taking priorities into account. The next sections explain the underlying architecture of the developed scheduler, detailing how this MIP model was created to obtain schedules that address the improvements mentioned here. They also describe the mechanisms and policies followed to detect opportunities and failures and how to manage the rescheduling processes they cause.

## Modeling the MIP Problem

The scheduler has a pool of remaining tasks with different parameters. It requires a model to describe the constraints that must be fulfilled to obtain a valid solution. The model works with the input data provided, which in this case is the task pool. The solution returned is a valid schedule, if any. The variables that the solver tries to optimize are the starting $s$ and ending $e$ time for each task. It also optimizes the auxiliary binary variable $Previous(a, b)$ to calculate the optimal path according to the location of the previous task. The rest of the parameters is fixed by the input data. The formalization of the model is explained here with the symbols, the binary predicates and the constraints used. It is important to note that the distance measures described are modeled as predictions of how much time the robot needs to go from one location to another. All needed distances are precalculated with the help of the knowledge base before running the solver.

**Positive integer parameters**:
$i$, $j$, $k$: Any task of the pool
$w^{min}$: Minimum start time
$w^{max}$: Maximum end time
$s$: Start time (variable)
$e$: Ending time (variable)
$o$: Operation time
$c$: Cooling down time
$p$: Priority value higher than 0
$l^s$: Starting location
$l^e$: Ending location
$d(a, b)$: Distance (time estimation) between $a$ and $b$

**Binary parameters**:
$Previous(i, j)$: Task $i$ starts just before $j$ (variable)
$Depends(j, i)$: Task $j$ must start after $i$

**Constraints**:
$w_i^{min} \leq s_i \leq w_i^{max} - o_i - d(l_i^s, l_i^e)$
$w_i^{min} + o_i + d(l_i^s, l_i^e) \leq e_i \leq w_i^{max}$
$Previous(i, j) \Rightarrow s_i < e_i < s_j$
$\neg Previous(i, j) \Rightarrow s_i < s_k < s_j$
$Previous(i, j) \Rightarrow e_j \geq s_j + o_j + d(l_i^e, l_j^s) + d(l_j^s, l_j^e)$
$Depends(j, i) \Rightarrow e_i < s_j$
$Depends(j, i) \Rightarrow c_j \geq e_j - e_i$

**Objective function**:
$$\text{Minimize} \sum_{i=1}^{n} e_i p_i$$

The first two constraints restrict the value that the start and ending time variables can have to ease the searching process, taking the operation time and distance into account. The third constraint avoids the overlapping of tasks by stating that the starting time of a task must be lower than its ending time and that the ending time of a previous task must be lower than the starting time of any other one that follows it. The fourth one states that a task $i$ cannot be previous of a task $j$ if there is any other task $k$ which starts between $i$ and $j$. The fifth determines that the ending time must be greater or equal than the starting time plus the operation time plus the time needed to travel between the ending location of the previous task and the starting location of the current one and plus the time to travel between the starting and end locations of the current task. The sixth one controls the dependence of a task by stating that a task that depends on another must start after the other has finished. The last one states that the cooling down time must be greater or equal than the time between the ending of its dependence and the ending of the dependent task.

The model has an objective function to optimize the solutions. This function is the sum of the ending time of each task multiplied by their priorities. This guides the search to reduce the total time span of the schedule, but also to execute tasks with high priority earlier. A dummy initial task to take the initial location of the robot into account is used to improve this optimization.

When all constraints are met, a valid schedule has been found. However, this solution will probably not be optimal. The scheduler will continue searching for an optimal solution during a certain time threshold. If the optimal schedule is found, the solver returns it to start the execution of the first task when needed. If the solver cannot find an optimal solution in the given time, it returns a suboptimal solution.

The solver sometimes detects when a schedule is unfeasible, but if the problem is too hard, the solver could continue searching forever. If the solver cannot find a suboptimal solution in the given time, the schedule is considered unfeasible. This leads to different policies that are explained in the next section.

Before starting the search, the solver also checks if the input data are valid to avoid wasting time searching for an impossible solution. All integer variables must be positive, and priority higher than 0. Additionally, it considers these additional conditions.

**Checks**:
$w_i^{min} + o_i + d(l_i^s.l_i^e) < w_i^{max}$
$c_i \geq o_i + d(l_i^s.l_i^e)$

These checks can rely only on parameters of the own task. That is why they do not include the distance between the ending location of the previous task and the starting of the current one. That has to be calculated by the solver.

## Monitoring

The Monitoring part receives partial states of the world from Execution, as shown in Figure 1. These states contain different elements like the current state of the task, the identity of the people near the robot or the objects placed in its basket. Monitoring receives this partial state and detects opportunities and failures for the task being executed, using the task definition (as the one shown in Table 1). Then, it decides whether it is necessary to run the MIP solver to take the possible generic and specific opportunities and failures into account and reschedule if needed. The outcome of this reschedule may cause the interruption of the current task. Monitoring sends the updated task pool to the MIP scheduler and receives a solution. Then it sends the first task to Execution.

There are two ways to add tasks to the pool. In the first one, the user sends a task to the robot through the user interface. The task is received, and the scheduler component tries to obtain a valid schedule. If it cannot, the new task is rejected, and the user must change the parameters of the task. In the second way, a task is generated internally by the robot in response to a detected opportunity. Tasks can be removed from the pool if a user aborts them or if they cannot be executed anymore because a failure. A task can finish earlier than expected due to a detected opportunity or a failure.

Opportunities can appear at any moment and may modify the current schedule. Generic internal parameter *Opportunities* of each task may indicate the existence of potential opportunities for the CoBot domain. Monitoring has a record of these parameters for all task and when something in the state of the world matches such parameters, a scheduled task can be modified or canceled, triggering then a rescheduling process. The outcome of this reschedule may cause the interruption of the present task. Currently all reschedules are done from scratch, without reusing previous solutions.

For example, DeliverDrink can have a person as a receiver. If the robot detects that person while it has the object in the basket, it can interrupt the current task to give the object to that person. This can happen with other tasks such as DeliverMessage and Escort. Similarly, if the robot has to give an object to someone and it detects that it already has one in its basket, the robot can deliver that object directly. In the same way, if the robot detects a very important person, Monitoring can add a new high priority task to the pool to reach him and perform a demonstration or give him information. The mechanism to determine if this new task can be scheduled or not is based in a gain metric explained in the next subsection.

Failures are similar to opportunities, except because they are unavoidable. Currently, CoBots control some lower-level failures like people not answering questions or not present to get some objects from the basket. Others like someone stealing an object from the basket that needs to be delivered, need the control of invariants along all the tasks of the pool. If an invariant parameter of a task changes in the state of the world (registered in the internal parameter *Failures*), the task is modified or even canceled, triggering a reschedule. Monitoring can also add tasks on failures to redo a chain of dependent tasks. Other possible failures are the need to charge the battery and reaching a certain delay threshold in which the planned starting time of the next task and the actual time is too much.

## Rescheduling Policy

In response to a failure, Monitoring can remove tasks from the pool if the solver cannot find a suitable plan while rescheduling (it cannot redo a task because there is not enough time, for example). In this case, priority and amplitude of the time window are important parameters to cancel a task or not. The tasks that now do not have enough time to be completed are canceled directly. If, after this, the solver cannot find a plan, Monitoring starts canceling the next task with the lowest priority and the smallest time window that overlaps another. This process will continue until a schedule is found. When a task ends unexpectedly, the owner is alerted by email.

In the case of opportunities, if a task ends earlier than expected, Monitoring also triggers a reschedule to take advantage of it. If a new task is added into the pool (talk with the spotted VIP, for instance) and the scheduler cannot find a plan, then Monitoring evaluates the situation. The metric to determine this is the gain value $g$ which is the sum of the priorities of the scheduled tasks.

**Gain**: $g = \sum_{i=1}^{n} p_i$

A VIP task has a very small-time window and very high priority. Canceling a DeliverDrink task to redo it later is a valid option if possible because more tasks can be done, and more gain will be obtained. The policy developed in this work tries to preserve the tasks already scheduled because it only considers redoing the current one. This could be problematic if there is a tight cooling-down time for the task, but subdivisions of tasks like DeliverDrink could relax the constraints by delaying the delivering of the drink if the cooling-down time is not reached.

For instance, in case that a VIP task appears just after a hot coffee is made and before delivering it, the scheduler has to consider all the cases explained in the Interrupting Tasks section. Basically, it tries to schedule everything by redoing the current subtask later (in the case of the drink in this example). If it cannot, it tries to redo the whole DeliverDrink task. If it is impossible, then it evaluates whether to cancel the current task or the new important task by using the gain measure. This means that the solver has to be called 1, 2 or 4 times, depending on the case.

Rescheduling can take some time, so although it can be done while the robot is traveling, opportunities may need a fast response to take advantage of them. In CoBot a reasonable time for each schedule could be 10 seconds, for example. With that threshold, we may need up to 40 seconds in the worst case scenario.

# Experiments

The goal of this section is to evaluate the rescheduling architecture as an effective way to organize, execute and monitor the tasks performed by the robot in a social environment. Therefore, it is required to determine whether the system is

fast enough to provide good schedules in a small amount of time. In this sense, a maximum running time of 10 seconds (under regular computing capabilities) is desired as the maximum time that the robot should be waiting to the solver. This evaluation separately describes two main aspects of this work[1]. The first one is the performance of the developed scheduler in terms of solving time and schedule quality. The second one highlights the behavior of the Monitoring part by using an example task pool and three feasible schedules. Using robot execution time (for instance, navigation time) to reduce decision delays (using that time to compute new schedules or improve current ones) is out of the scope of this work.

## MIP scheduler

The external MIP solver used is GLPSOL[2]. To cover the temporal requirement specified above, the solver permits to limit the time used to find a solution in several ways. The experiments performed are focused in two of them: limiting the maximum solving time and accepting a solution when a certain quality measure is reached. Solvers provide the relative MIP gap percentage, which gives an indirect insight of the quality of a solution. In essence, this measure is related to the current numerical upper and lower bounds, so when it reaches 0 % the solution is proven optimal. However, for clarity purposes, these experiments also use a function to represent quality ($q$) more directly:

$$q = (\sum_{i=1}^{n} e_i)/n$$

Quality is the sum of the end times for each task ($e_i$) divided by the number of tasks in the pool ($n$). The number by itself is meaningless, but when compared, it gives the average amount of time that a task will be delayed while using a configuration of the solver instead of another. It does not use priorities, but it is enough for the purposes of this section.

The experiments evaluate three configurations: 1) Solving time limited to 10 seconds with 4.4 % of relative MIP gap tolerance, 2) 10 seconds limit without tolerance and 3) 30 seconds limit. The memory usage is always under 100 MiB so there is no need to limit it too. The solver uses a base set of 480 random instances of task pools (set A). This set has some unfeasible or too hard instances so, to make some comparisons, the results also refer to subsets B and C which are contained in A. Subset B has 12 random instances per each size of the task pool, ranging from 1 to 15 (180 instances). They have been solved in all three configurations. Subset C is the same as B except that the ranges of pool sizes are limited from 8 to 15 (96 instances) to focus on the hardest ones.

The 4.4 % tolerance value used in the first configuration is the average relative MIP gap reached in the second configuration. Figure 2 uses the set B to show the average solving times for each task pool size and configuration, where instances have at least one solution in all cases.

[1]All experiments were carried out in an Intel Core i3-2330M CPU at 2.2 GHz with 4 GiB of RAM.

[2]http://www.gnu.org/software/glpk

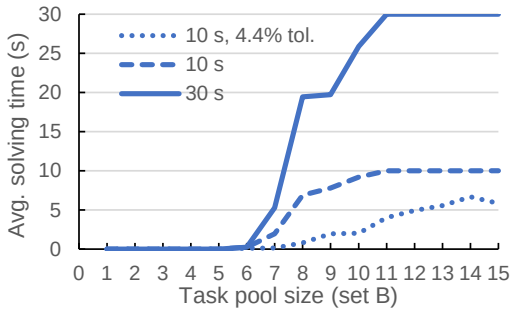



Figure 2: Evolution of the average solving time along different task pool sizes (set B ⊂ A).

The solver starts using the whole time allowed when the pool has 11 tasks in it, except when there is a MIP gap tolerance of 4.4 %. In this case, the average maximum solving time is under 8 seconds. That indicates that a valid solution is found fast and the rest of the time is used to optimize it or prove its optimality. The gap tolerance could be useful if the impact on the quality of the solution is acceptable for the intended application. Figure 3 shows the qualities for the set B. Time is measured in minutes.

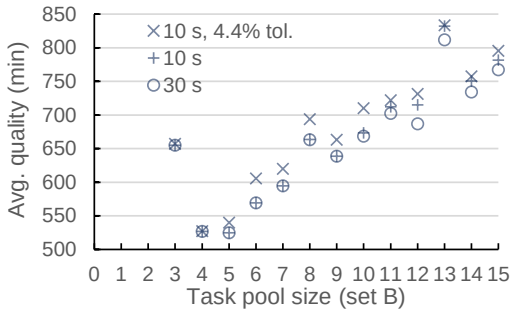



Figure 3: Average quality of the planned schedules with different pool sizes (set B ⊂ A). The Y axis starts at 500 to improve readability, but instances for sizes lower than 3 are hidden below.

Spending 30 seconds to plan the schedule is usually much better, in terms of quality, than using only 10 seconds with tolerance at 4.4 %. However, when there are 10 tasks or fewer, 10 seconds seem enough. CoBot should start responding in 10 seconds at most, although the results show that spending some seconds more can save much time. Table 2 details the 6 types of outcomes that the solver can return.

For the full set A of 480 instances, only 0.8 % of the instances passed the checks and were proven unfeasible. Also, only 11 % of the instances reached the time threshold without being solved. Sets B and C were solved by all configurations. Few optimal solutions were found from 10 to 30 seconds, but the average quality increased especially in the set C of difficult instances. Using the gap tolerance reduces the solving time sensibly but also the quality of the solutions. From 10 to 30 there is a reduction of 12 minutes of delay per task in set C. All these results suggest that the relative

MIP gap tolerance could be tuned to find a feasible schedule very fast to start responding as soon as possible, and then continue refining the schedule while the robot is working or waiting to start the next scheduled task.

| | Configuration | 10 s, 4.4% tol. | 10 s | 30 s |
|---|---|---|---|---|
| *Set A* | Time out: no solut. | 11.0% | 11.0% | 8.8% |
| | Proven unfeasible | 0.8% | 0.8% | 0.8% |
| | Check failed | 4.4% | 4.4% | 4.4% |
| | Proven optimal | 16.3% | 42.7% | 43.1% |
| | Min. gap reached | 54.0% | 0.0% | 0.0% |
| | Time out: found | 13.5% | 41.0% | 42.9% |
| | Solutions found | 83.8% | 83.8% | 86.0% |
| *Set B* | Proven optimal | 17.8% | 51.1% | 52.2% |
| | Min. gap reached | 68.3% | 0.0% | 0.0% |
| | Time out: found | 13.9% | 48.9% | 47.8% |
| | Av. solver time (s) | 2.14 ± 3.6 | 5.07 ± 4.9 | 14.7 ± 14.8 |
| | Av. quality (min) | 611 ± 256 | 596 ± 250 | 590 ± 247 |
| *Set C* | Proven optimal | 0.0% | 10.4% | 12.5% |
| | Min. gap reached | 74.0% | 0.0% | 0.0% |
| | Time out: found | 26.0% | 89.6% | 87.5% |
| | Av. solver time (s) | 3.98 ± 4.1 | 9.24 ± 2.5 | 26.88 ± 8.6 |
| | Av. quality (min) | 738 ± 135 | 721 ± 137 | 709 ± 137 |

Table 2: Solution type for the three configurations. Set C ⊂ B ⊂ A. Subsets B and C also have average solving times and qualities.

## Monitoring

The target of this section is to evaluate how the rescheduling module works when some unexpected event occurs. Table 3 illustrates three possible schedules after the arrival of a VIP at minute 20. There are two kinds of tasks in each schedule: three *Deliver coffee* (C1, C2, C3) for the same person and one *Show off* (VIP). *Deliver coffee* is subdivided in two tasks that must be executed in order. All tasks have priority 1 except VIP, which has 100. The cooling down time for the coffee is 15 minutes. The time to travel from the initial location is 4 minutes. The time to travel from the coffee maker to the delivering location is 3. The VIP is in the same location of the coffee maker.

These schedules could be planned applying the described rescheduling policies in less than 10 seconds. This was because the pool is small enough and VIP can be scheduled before or after C1b, so there is no need to run the solver more times. Currently the system schedules from scratch when a new task arrives, so the results of the previous subsection are valid also in this one.

As it can be seen, the scheduler has locations into account, changing the duration of the tasks if the robot has to move to another place to perform them. Because of this, the scheduler tries to join all first parts and all second parts of *Deliver coffee* tasks to save trips. This is a clear improvement over the original scheduler. In schedule A, it is only possible to join two of them because the cooling down value of 15 minutes is too small.

Schedule A

| Task | Start | End |
|---|---|---|
| … | 0 | 10 |
| C1a | 11 | 20 |
| C2a | 21 | 26 |
| C1b | 27 | 31 |
| C2b | 32 | 33 |
| C3a | 34 | 42 |
| C3b | 43 | 47 |
| VIP | 48 | 53 |
| **Cost** | 739 | |

Schedule B

| Task | Start | End |
|---|---|---|
| … | 0 | 10 |
| C1a | 11 | 20 |
| C2a | 21 | 26 |
| C1b | 27 | 31 |
| C2b | 32 | 33 |
| VIP | 34 | 39 |
| C3a | 40 | 45 |
| C3b | 46 | 50 |
| **Cost** | 605 | |

Schedule C

| Task | Start | End |
|---|---|---|
| … | 0 | 10 |
| C1a | 11 | 20 |
| VIP | 21 | 23 |
| C2a | 24 | 29 |
| C1b | 30 | 34 |
| C2b | 35 | 36 |
| C3a | 37 | 45 |
| C3b | 46 | 50 |
| **Cost** | 454 | |

Table 3: Valid schedule examples after the arrival of a new task VIP at time 20. Units expressed in minutes.

The VIP task is performed at different positions in each table and this has an effect in the total time of the schedule and the cost that the scheduler has to minimize. Schedule A is the worst because the important task is executed later, and it takes 3 minutes more than the others because it needs an extra trip. Schedule B is much better because it saves one trip and executes the important task earlier. Schedule C executes the VIP task as soon as it arrives because the cooling down time is wide enough to perform VIP and C2a between C1a and C1b. All three schedules are valid because the only hard constraint is that VIP must start at or after minute 20. To execute the most important tasks first is a soft constraint.

As shown in the previous section of the experimentation, in scenarios with constrained task pools the optimal plan is not always obtained because of the need of fast responses. Sometimes false negatives occur (solutions not found by the solver that exist), which is undesirable especially when rescheduling a new task like VIP. However, in practice, the monitoring part works as intended because of the low planning time limit and because normally there are few remaining tasks in the task pool. There are also some online videos to watch the new scheduler at work[3].

## Conclusion

This manuscript presents a new architecture of task execution, monitoring and rescheduling, which was developed in top of the current CoBots to improve their capabilities. Apart from the scheduling abilities of the original approach (Coltin, Veloso, and Ventura 2011), the new one is focused on fast solving times to ease the human-robot interaction, opportunity and failure detection, rescheduling with task interruptions, priority optimization and dependent tasks with cooling down times (a hot coffee cannot be delivered much later after being done). The developed component is generic enough to be applied in other multilevel architectures like NAOTherapist (González, Pulido, and Fernández 2017) by changing only the task specification and the communication interfaces. The experimentation shows that the system can perform well in normal conditions, although in scenarios with numerous remaining tasks the quality of the

[3] http://goo.gl/r2cszx

obtained schedules can be affected, especially if the allowed solving times are low.

Future work may include deeper integration with the current architecture to take control of some aspects that are managed in a lower level. The policies to take advantage of opportunities could also be improved by trying a quicker version of the VIP task, for example, in difficult task pools. This could lead in more reschedules, but they could be done while the robot is working. Also, alternative contingent plans could be considered, for example, to give a coke instead of a coffee because it is better than nothing. This would use more the concept of gain.

## References


Alcázar, V.; Guzmán, C.; Prior, D.; Borrajo, D.; Castillo, L.; and Onaindia, E. 2010. PELEA: Planning, Learning and Execution Architecture. In *Proceedings of the 28th Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG)*.

Cashmore, M.; Fox, M.; Long, D.; Magazzeni, D.; and Ridder, B. 2017. Opportunistic Planning in Autonomous Underwater Missions. *IEEE Transactions on Automation Science and Engineering (T-ASE)* PP(99):1–12.

Chen, D.-S.; Batson, R. G.; and Dang, Y. 2010. *Applied Integer Programming: Modeling and Solution*. John Wiley & Sons.

Coltin, B.; Veloso, M. M.; and Ventura, R. 2011. Dynamic user task scheduling for mobile robots. In *Automated Action Planning for Autonomous Mobile Robots, Papers from the 2011 AAAI Workshop, San Francisco, California, USA*.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory & Practice*. Elsevier.

Ghallab, M.; Nau, D.; and Traverso, P. 2014. The Actor's View of Automated Planning and Acting: A Position Paper. *Artificial Intelligence (AIJ)* 208:1–17.

González, J. C.; Pulido, J. C.; and Fernández, F. 2017. A three-layer planning architecture for the autonomous control of rehabilitation therapies based on social robots. *Cognitive Systems Research (CSR)* 43:232–249.

Guzmán, C.; Castejón, P.; Onaindia, E.; and Frank, J. 2015. Reactive execution for solving plan failures in planning control applications. *Integrated Computer-Aided Engineering (ICAE)* 22(4):343–360.

Rosenthal, S., and Veloso, M. 2012. Mobile Robot Planning to Seek Help with Spatially-Situated Tasks. In *Proceedings of AAAI'12, the Twenty-Sixth AAAI Conference on Artificial Intelligence*.

Schermerhorn, P.; Benton, J.; Scheutz, M.; Talamadupula, K.; and Kambhampati, S. 2009. Finding and Exploiting Goal Opportunities in Real-Time During Plan Execution. In *Proceedings of the 21st International Conference on Intelligent Robots and Systems (IROS)*, 3912–3917.

Veloso, M.; Biswas, J.; Coltin, B.; and Rosenthal, S. 2015. CoBots: Robust Symbiotic Autonomous Mobile Service Robots. In *Proceedings of IJCAI'15, the International Joint Conference on Artificial Intelligence*.

# Trade-offs Between Communication, Rescheduling, and Success Rate in Uncertain Multi-Agent Schedules

**David A. Chu** and **Grace Diehl** and **Marina Knittel** and **Judy Lin** and **Liam Lloyd** and **James C. Boerkoel Jr.**
Computer Science Department
Harvey Mudd College
Claremont, California 91711
{dchulasso, gdiehl, mknittel, julin, wlloyd, boerkoel}@hmc.edu

**Jeremy Frank**
NASA Ames Research Center
Mountain View, California 94035
jeremy.d.frank@nasa.gov

### Abstract

Generating and executing multi-agent schedules is difficult in uncertain environments. The current state-of-the-art algorithm maintains a high success rate by rescheduling frequently, but this approach involves substantial resource overhead due to computing and communicating new schedules. Aggressive rescheduling could thus reduce overall mission duration in situations where agents have limited energy and computing power. We thus explore the trade-off between the number of reschedules and success rate. Specifically, we propose three new algorithms that strategically decide when rescheduling is most likely to meaningfully increase the probability of success. Additionally, we empirically show that, while there is a trade-off between the number of reschedules and schedule success rate, it is possible to reduce the number of reschedules without proportionally decreasing success. We find that one of our approaches, Allowable Risk, allows us to gracefully trade reductions in success rate for significant reductions in the number of reschedules, and thus communication, of a state-of-the-art dynamic scheduling algorithm.

## Introduction

Generating and executing in multi-agent systems is an enabling technology for many applications, such as cooperative teams of airborne, surface operating, or underwater robots. Providing this capability requires effective multiagent coordination, since these applications involve uncertain environments that may challenge the success of a mission. In this paper, we focus on a scenario where this requirement is met by scheduling centrally and broadcasting a joint schedule to individual agents. However, a fixed, predefined schedule only uses the information that was available when it was created. As the mission progresses, uncertain events (e.g. unexpectedly long task durations) may disrupt the schedule. To take advantage of this new information, we need an algorithm to reschedule in response to these uncertain events.

*Dynamic execution algorithms* reschedule in response to new information, potentially increasing the chance that the mission succeeds. However, they may reschedule frequently, which means that the centralized scheduler would have to frequently send out new schedules to agents. While this frequent communication would not be a problem in some circumstances, in many applications, conserving battery power is an issue and communications are energy intensive, so any extra communication detracts from the time agents can spend completing their mission. To address this problem, we propose three new execution algorithms that limit how often schedules are sent:

- *Sufficient Improvement* sends out a new schedule only if it is likely to significantly increase the predicted probability of success;
- *Allowable Risk* reschedules more often when schedules are more risky and less often when schedules are less risky; and,
- *Coordination Targeting* uses constraints between agents to determine when to reschedule.

We conduct an empirical evaluation showing that Sufficient Improvement and Allowable Risk can decrease rescheduling without proportionally decreasing success rate, and evaluate the trade-offs between rescheduling frequency and success rate. We find that Allowable Risk allows us to gracefully trade reductions in success rate for significant reductions in the number of reschedules, and thus communication, of a state-of-the-art dynamic scheduling algorithm.

## Background

Unmanned aerial vehicles (UAVs) have been used for missions ranging from collecting data on wildlife to monitoring wildfires. More complex missions have been proposed and, on small scales, demonstrated using UAV teams that communicate over radio cross-links with each other (e.g. Cesare et al.; Li et al. (2015; 2016)). Despite numerous advances in energy storage, battery life generally limits UAV mission duration (Quach et al. 2013). In particular, communications between UAVs and between the UAV and base stations can consume considerable energy, spurring research in energy-efficient communications between UAVs to enable relays (Zhang, Zheng, and Zheng 2017). In a UAV team, conducting missions in the presence of uncertainty that may require

rescheduling, communicating the new schedule consumes energy, and could therefore shorten the mission.

Let us consider a wildfire surveillance coordination problem involving two UAVs, Agent A and Agent B. Both agents must capture infrared images of the wildfire from different locations and send them back to a base station. Suppose for the entirety of this example that Agent B is between Agent A and the base station, and can communicate with both, but Agent A can only communicate with Agent B. Agent A takes an infrared image, but must subsequently relocate to a safe position due to unsafe flying conditions in its vicinity. Once Agent A is in its new position, it must send the image to Agent B, which can later send the image to the base station. However, Agent B must also travel a (shorter) distance before relaying Agent A's image, because it must take a second image at the new location. Agent B's second image acquisition must take place immediately after its first image, and thus before receiving Agent A's communication. Agent A's and Agent B's navigation tasks take around 40 and 10 seconds, respectively, and these durations are uncertain due to wind and localization error. The image sending/receiving task also takes an uncertain amount of time–around 5 seconds. Because the agents would have to do much more than mentioned (i.e. this subproblem is a small portion of a much larger problem), we want the agents to finish their tasks within a specified amount of time, namely 60 seconds.

In general, this class of problem can be posed and solved as a Decentralized Partially Observable Markov Decision Problem (DEC-POMDP). For instance, Wu, Zilberstein, and Chen (2011) consider how to reschedule in such cases in the presence of little communication. For our work, we limit the expressiveness of the problem under consideration to Probabilistic Simple Temporal Networks (PSTNs), defined in the next section. While tractable, the PSTN framework still exposes the fundamental problem of deciding when to reschedule, and how to communicate changes of schedules between agents.

## Probabilistic Simple Temporal Networks

A *Simple Temporal Network* (STN), $S = (T, C)$, consists of a set $T = \{t_0, t_1, \ldots, t_n\}$, where each timepoint $t_i$ represents the time at which a distinct event happens, and a set $C$ of binary constraints $c_{ij}$ on events in $T$. These constraints are of the form $t_j - t_i \leq b_{ij}$, for some $b_{ij} \in \mathbb{R}$ (Dechter, Meiri, and Pearl 1991). The two constraints between $t_i$ and $t_j$ can be written concisely as $t_j - t_i \in [-b_{ji}, b_{ij}]$. STNs are often encoded as directed graphs, where events are vertices and constraints are edges. A *schedule* is an assignment of values to events such that all constraints are satisfied. An $STN$ is considered *consistent* if it has at least one schedule.

Since the physical world is inherently uncertain, accounting for uncertainty in our representation allows it to better model real-world problems. In a *Simple Temporal Network with Uncertainty* (STNU), the set of constraints $C$ is divided into two disjoint subsets, called $C_R$, the set of *requirement* edges, and $C_C$, the set of *contingent* edges. Requirement edges are identical to the constraint edges in a standard STN. A contingent edge, however, represents that the time that elapses from $t_i$ to $t_j$, given by $\beta_{ij} \in [-b_{ji}, b_{ij}]$, is chosen by an uncontrollable process and is unknown prior to execution (Vidal and Ghallab 1996).

An event whose incoming edges are all requirement edges is known as an *executable* timepoint, because the agent executing the schedule controls when it happens, or *executes*. A timepoint with an incoming contingent edge is known as a *contingent* timepoint, since it happens automatically some time after the timepoint that initiates the contingent edge. When a contingent timepoint happens it is said to be *received*. We call the set of contingent timepoints $T_C$, and the set of executable timepoints $T_X$.

STNUs are *strongly controllable* if each executable timepoint can be restricted such that, for *all* possible contingent timepoint outcomes, *all* requirement constraints are satisfied. Not all STNUs satisfy this restrictive property. Some STNUs are *dynamically controllable*, which means a limited form of contingent schedule can be produced prior to execution; these schedules describe, in a compact way, when to schedule future executable timepoints in response to contingent timepoint outcomes. In this paper, we use dynamic rescheduling to address these uncertain outcomes.

A *Probabilistic Simple Temporal Network* (PSTN) extends an STNU by adding information about the uncertain processes that govern contingent edges. For a PSTN's contingent edges, the time that elapses from $t_i$ to $t_j$ is chosen by a random variable $X_{ij}$, whose value is determined at execution by some PDF $P_{ij}$ (Tsamardinos 2002; Brooks et al. 2015). Since contingent edges in PSTNs are governed by unbounded probability distributions, they cannot be strongly controllable. However, as we will later discuss, some algorithms still use the idea of strong controllability to solve PSTNs.

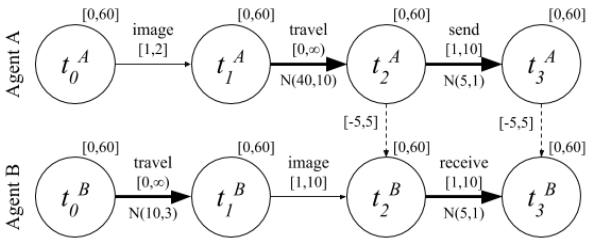



Figure 1: PSTN for our example problem.

**Example Problem** The PSTN representation of our running UAV example problem is shown graphically in Figure 1. Each vertex represents a timepoint. The agent to which a particular timepoint corresponds is indicated by the superscript. Two such timepoints could be the start and end times of Agent A's image taking task, which are represented as $t_0^A$ and $t_1^A$, respectively. Directed edges represent temporal constraints and are labeled with the range of time that is allowed to elapse between the occurrence of the events represented by the source and target timepoints. There are three types of directed edges: thick edges represent contingent edges, dashed edges represent interagent constraints, and straight, slim edges represent requirement edges. All tasks must be completed within 60 seconds, shown by the constraint [0,60] above each vertex. Agent A's subproblem is contained in the top half of the figure, while Agent B's subproblem is contained in the bottom half of the figure.

## Execution Algorithms for PSTNs

**Early First** Early First is a naïve algorithm for deciding when to execute the timepoints in a PSTN. As its name implies, it executes timepoints as soon as they can be executed–when they are both *live*, meaning that they are within their acceptable time range, and *enabled*, meaning that all predecessor timepoints have been executed.

**The Static Robust Execution Algorithm** While algorithms like Early First can be effective in practice, they are agnostic as to the impact of uncertainty on performance. In our UAV example problem, if both agents start navigating as soon as possible, it is highly likely that Agent B will arrive at its destination more than 10 seconds before Agent A, resulting in failure. To maximize the probability of success, Agent B should wait before navigating. The Static Robust Execution Algorithm (SREA) was motivated by this limitation (Lund et al. 2017). SREA tries to address this limitation by maximizing *robustness*, the probability that all events are executed without violating constraints (Brooks et al. 2015). Robustness is the complement of risk, introduced in (Fang, Yu, and Williams 2014).

In order to maximize robustness, SREA attempts to create a strongly controllable STNU with a minimum probability of failure. SREA sets a maximum probability $\alpha$ that a contingent edge in the original PSTN fails because it is too short or too long. This makes $1 - \alpha$ the minimum probability mass of the contingent edge captured by a corresponding interval in the STNU. To find the optimally robust schedule, SREA does a binary search over $\alpha$. For each $\alpha$, it uses a linear program to maximize the probability mass captured by the interval over each contingent timepoint. In a sense, SREA maximizes the probability that uncertain events will occur during these intervals. Once it finds the optimally robust schedule, it can execute this more constrained schedule using Early First. In our running example, SREA would constrain Agent B to wait before navigating to maximize the probability that the arrival times of both agents overlap.

SREA is good at using initial information to maximize the probability of success. However, it can fail when uncertain timepoints fall outside of their designated intervals during execution. In addition, SREA is limited because it cannot re-optimize constraints when new real-time information, such as the actual time of an uncertain event, is gained. This is because SREA is a *static* algorithm, in that it does not change the schedule in real-time. Therefore, dynamically updating the guiding schedul can be beneficial in situations with uncertain events.

**The Dynamic Robust Execution Algorithm** The Dynamic Robust Execution Algorithm (DREA) builds on SREA with the goal of maximizing robustness by adding the ability to incorporate new information during execution Lund et al. (2017). It creates an initial schedule by running SREA and uses it to guide execution. Whenever a contingent timepoint is received or enabled, DREA updates the PSTN with this new information, and calls SREA to create a new schedule.

## Reducing Communication in DREA

DREA has a significantly higher success rate than Early First on the set of benchmark PSTNs in Lund et al. (2017). However, this high success rate comes with the cost of large amounts of rescheduling. In many scenarios, including as our UAV example, communicating new schedules is costly, so DREA may have undesireably high computational overhead.

With this in mind, we have investigated three possible methods for limiting when DREA reschedules such that communication is reduced without drastically diminishing success rate. *Sufficient Improvement* seeks to reduce rescheduling frequency by only rescheduling when the new schedule has a better predicted probability of success than the previous one. *Allowable Risk* makes rescheduling frequency proportional to the quantity of risk present in the problem. *Coordination Targeting* attempts to focus rescheduling only in preparation for events subject to inter-agent constraints.

Our algorithms are modifications to DREA. They act exactly as DREA does except when a timepoint is enabled or received, in which case, instead of always rescheduling, they more judiciously decide whether to reschedule. Sufficient Improvement and Allowable Risk use the minimum improvement thresholds $m$ and $x$, respectively, in their calculations to make this decision. Allowable Risk also uses a counter $k$ that increments whenever a timepoint is received and resets when rescheduling. The value of this counter and the threshold inputs is explained in our algorithm details. We rewrite DREA as a routine that utilizes our subroutines in Algorithm 1.

**Algorithm 1:** Modified DREA

```
Input : A PSTN S, a rescheduling strategy s
Input : A min improvement threshold 0 < m < 1
Input : A min success threshold 0 < x < 1
guideSTN, α ← SREA(S) ;
k ← 0 ;
while S.isConsistent() and not S.allExecuted() do
    if any t ∈ T_C is received or enabled then
        S.update(t);
        if t is received then
            k ← k + 1 ;
        (guideSTN, α, k) ←
          maybeReschedule(guideSTN, s, m, x, α, k)
    else
        foreach live & enabled t ∈ T_X according to
          guideSTN do
            S.execute(t);
            guideSTN.execute(t);
```

## Sufficient Improvement

The concept underlying Sufficient Improvement (SI) is to only use schedules that sufficiently improve the chances of success. Like DREA, it will always create a new schedule any time it gets new information. Unlike DREA, SI will only send out this new schedule if it calculates that the new schedule has a significantly larger probability of success than the schedule currently in use.

We represent SI as part of the $maybeReschedule$ subroutine of DREA presented as Algorithm 2. The

$maybeReschedule$ subroutine is called whenever a time point is received or enabled. SI requires an input minimum threshold for improvement $0 < m < 1$. When we run SI, it first calculates a new potential schedule with SREA, then records the number of contingent events left in our STN. SI then uses these values to determine if the difference between the probability of success of the new schedule and the probability of success as calculated at the last rescheduled is sufficient. When DREA runs SREA to generate a schedule, as noted above, it creates an interval around each uncertain edge that captures some amount of the probability mass of that edge. For each duration, this interval captures at least $1 - \alpha$ of the probability mass. SI calculates the probability of a schedule executing without violating a constraint by multiplying the captured probability mass for each uncertain edge. Generally speaking, this probability is bounded by $p \leftarrow (1-\alpha)^n$, where $n$ is the number of uncertain edges. Note that this is actually an underestimate, since SREA expands the intervals somewhat beyond $1 - \alpha$ of the probability mass. SI then takes the difference between the new probability and the old probability. If this difference is above our threshold value $m$, it uses the new schedule.

If a contingent edge invalidates the current schedule generated by SREA by falling outside of its allowed range, SI continues to use the now-violated schedule until rescheduling is triggered. The execution is not considered failed unless one of the constraints from the original STN is violated. All of our rescheduling strategies handle violations of the SREA-generated schedule in this fashion.

**Algorithm 2:** $maybeReschedule()$ Subroutine

**Input :** An STNU $guideSTN$
**Input :** A rescheduling strategy $s$
**Input :** A min robustness $\alpha$
**Input :** A received contingent event count $k$
**Input :** A min improvement threshold $0 < m < 1$
**Input :** A min success threshold $0 < x < 1$
**if** $s ==$ "$SI$" **then**
  $(maybeGuideSTN, \alpha_1) \leftarrow \text{SREA}(S)$ ;
  $n \leftarrow maybeGuideSTN.numCEventsLeft()$ ;
  $p_0 \leftarrow (1-\alpha)^n$ ;
  $p_1 \leftarrow (1-\alpha_1)^n$ ;
  **if** $p_1 - p_0 > m$ **then**
    $guideSTN \leftarrow maybeGuideSTN$ ;
    $\alpha \leftarrow \alpha_1$ ;
  **return** $(guideSTN, \alpha, k)$
**else if** $s ==$ "$AR$" **then**
  $n \leftarrow 0$ ;
  **while** $((1-\alpha)^{n+1} > x)$ **do**
    $n \leftarrow n + 1$ ;
  **if** $k \geq n$ **then**
    $(guideSTN, \alpha) \leftarrow \text{SREA}(S)$ ;
    $k \leftarrow 0$ ;
  **return** $(guideSTN, \alpha, k)$

SI's threshold influences how often the algorithm reschedules. If the threshold is low, we expect the algorithm will reschedule often, resembling DREA. Alternatively, if the threshold is high, the algorithm will reschedule less often, performing like SREA. In between extreme values, we expect to see a trade-off, where we reduce communication but also decrease the success rate as we increase the threshold.

## Allowable Risk

Our next rescheduling strategy reduces rescheduling depending on the riskiness of a schedule. Allowable Risk (AR) decides how many uncertain events it can allow to occur statically with an acceptably high probability of success, as defined by a threshold. Then it simply allows those events to occur without intervening, and reschedules when they have all happened. This strategy limits communication by setting the rescheduling frequency in direct proportion to the risk associated with the intervals generated by SREA, since lower risk will allow larger groups of uncertain events to execute statically.

AR is also designed as part of the $maybeReschedule$ subroutine of DREA in Algorithm 2. It requires an input $0 < x < 1$ to represent the minimum robustness threshold. AR first finds the largest integer value of $n$ such that $(1-\alpha)^n > x$ for threshold $x$. AR then reschedules if the received event counter $k$ exceeds $n$ (see Alg. 1). When it reschedules, AR resets $k$ to 0. Rescheduling will also produce a new $\alpha$, leading to a new value for $n$ (in the **while** loop of Alg. 2). Thus, if SREA generates a schedule with high $\alpha$, AR will reschedule sooner than if SREA had generated a schedule with low $\alpha$. As a proxy for probability of failure, $\alpha$ ties rescheduling frequency to the risk present in the schedules generated by SREA.

Since AR treats a threshold as a minimum probability of success allowed, a higher threshold could lead to frequent rescheduling, resembling DREA. However, a lower threshold could lead to low success rate from too little rescheduling, performing like SREA.

## Coordination Targeting

Our last rescheduling strategy seeks to use the structure of STNs to identify occasions when rescheduling will be most meaningful. Intuitively, it seems likely that rescheduling in preparation for constraints between events belonging to different agents, or interagent constraints, will be particularly impactful, because these constraints represent more complex interactions involving multiple agents.

Based on this observation, we design Coordination Targeting (CT) to reschedule right before we execute executable events that are subject to interagent constraints, and right before we execute the last executable events preceding contingent events that are subject to interagent constraints. First, it sets a flag to track whether a timepoint has been received or enabled since last rescheduling. Next, if any executable timepoints are enabled, it creates a copy of the current PSTN with all requirement edges removed. The algorithm then checks whether any timepoints involved in interagent constraints are reachable from the currently enabled executable timepoints. If that condition is true and the flag is set to true, then CT calls SREA to generate a new schedule, and sets the flag to false. This way, we are able to focus rescheduling around the decisions we expect to matter the most: the final executable timepoints before an interagent constraint.

To further explain CT, consider running CT on the example PSTN illustrated in Figure 1. For the interagent constraint $t_2^B - t_2^A \in [-5, 5]$ the algorithm reschedules right before $t_1^A$ and right before $t_2^B$, since $t_2^B$ is an executable timepoint and $t_1^A$ is the last executable before $t_2^A$, a contingent timepoint. For the interagent constraint $t_3^B - t_3^A \in [-5, 5]$, the algorithm provides no additional rescheduling, because there are no executable timepoints between $t_2^A$ and $t_3^A$, nor between $t_2^B$ and $t_3^B$. In other words, rescheduling at this point would not be useful, since we don't get to make any new decisions before $t_3^A$ and $t_3^B$ are received.

This algorithm could fail to reduce rescheduling if all events are constrained by interagent constraints. If there are not enough events under interagent constraints, this algorithm may act similarly to SREA, and therefore have lower robustness. Unlike our other two algorithms, the limitations of this algorithm are entirely dependent on the input. Thus, this algorithm cannot be tuned.

**Algorithm 3:** Coordination Targeting

```
Input : A PSTN S
guideSTN ← SREA(S) ;
while S.isConsistent() and not S.allExecuted() do
    new ← False ;
    resched ← False ;
    foreach t received or enabled ∈ T_C do
        S.update(t) ;
        new ← True ;
    foreach t enabled ∈ T_X do
        if t is controllable then
            pGraph ← guideSTN.justCEdges ;
            foreach interagent constrained s ∈ T do
                if t.reachs(s, pGraph) and new then
                    resched ← True
    if resched then
        guideSTN ← SREA(S) ;
        new ← False ;
    foreach live & enabled t ∈ T_X according to
     guideSTN do
        S.execute(t);
        guideSTN.execute(t);
```

## Experimental Setup

Next we describe our experimental setup for empirically evaluating our approaches.

### Scheduling Problem Testbed

Our evaluation methods attempt to recreate the experiments of Lund et al. (2017) to yield comparable results. First, our data set contains the same PSTNs used to evaluate DREA in Lund et al. (2017). These PSTNs were generated by the random robot navigation problem generator of Brooks et al. (2015). They generally have 20 timepoint variables divided among 2 to 4 agents, 20 to 35 total constraints, and a maximum of 15 contingent edges. We generated 30 PSTNs per combination of input features, totaling to 1620 schedules.

To evaluate a wide range of problems, our PSTNs were constructed from varying three input features: *degree of synchronization*, *interagent constraint density*, and *standard deviation*. The first feature is *degree of synchronization*, which sets a time upper bound between any two constrained events for different agents. In other words, it is the "tightness" of bounds on interagent constraints. From the Lund et al. (2017) data set, degree of synchrony varies between 1000, 2000, and 4000 times the standard deviation (described later in this section).

Another input feature is *interagent constraint density*, which defines the fraction of requirement constraints that are between agents. Varying our results across different values for interagent constraint densities reveals the impact of agent coupling intensity on the success of our algorithms. Interagent constraint density is set to 0.4 or 0.8.

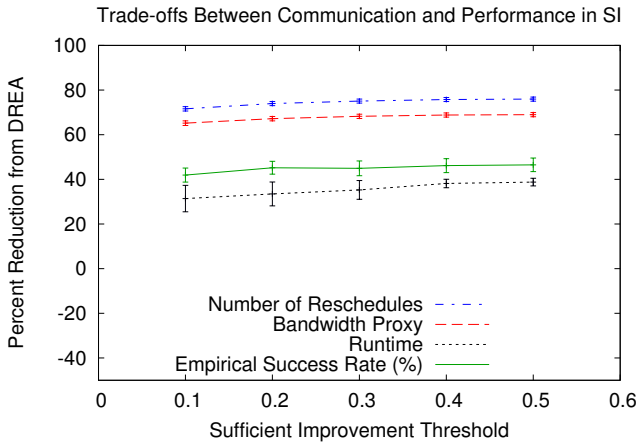
The last input feature is *standard deviation* of uncertain events to measure the degree of uncertainty in the input PSTN. The higher the degree of uncertainty, the wider the constraint probability distributions, and the harder it is for SREA to obtain a large $\alpha$ value. Varying the degree of uncertainty informs our understanding of our execution algorithms' performance since they aim to find a successful schedule in uncertain situations. The standard deviation was measured by a kurtosis metric for contingent edge distributions, which took values 1, 3, and 5. Kurtosis is a measure of how "peaky" a distribution is. A kurtosis value of 3 corresponds to a normal distribution, while a kurtosis of 1 corresponds to a flatter distribution with higher standard deviation, and a kurtosis of 5 corresponds to a more "peaky" distribution with lower standard deviation.

These problem features may generally impact the trade-off between robustness and communication that our algorithms are designed to manage. If we find our algorithms behave consistently across different values of these three input features, we will have a stronger basis to claim that our results generalize to different types of input scenarios. On the other hand, if our results are inconsistent across different values of one feature, we gain more insights into the scenarios in which one algorithm outperforms another.

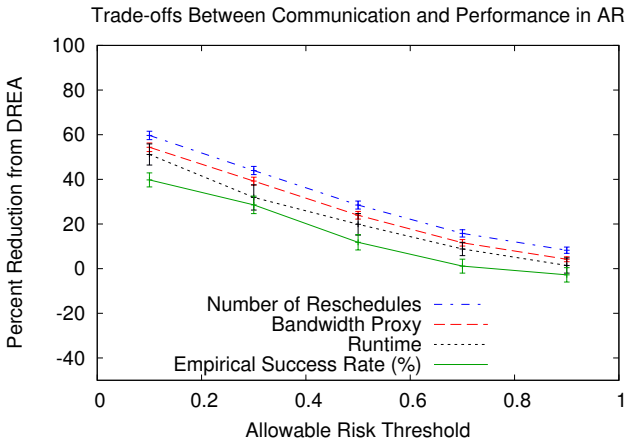
### Simulation

We also adapted the simulation software used in Lund et al. (2017). The simulation uses our data set to measure average success rate and duration. We additionally measured two other metrics, number of reschedules and accumulated bandwidth, to analyze trade-offs. The number of reschedules counts how often an algorithm sends out a new schedule, and accumulated bandwidth sums the sizes of all sent schedules, where the sizes are calculated as the sum of the number of edges and number of timepoints in the schedule.

We evaluate each algorithm on each PSTN in the data set 390 times, sampling uncertainty distributions during execution. For SI, we evaluate thresholds from 0.1 to 0.5 in steps of 0.1. For AR, we vary the threshold from 0.1 to 0.9 in steps of 0.2. In varying the thresholds, we gain more insight into the trade-offs between relaxed rescheduling constraints and schedule success rate.




(a) Sufficient Improvement (SI) threshold variation results




(b) Allowable Risk (AR) threshold variation results

Figure 2: Simulated results for Sufficient Improvement (SI) and Allowable Risk (AR). We plot the success rate, number of reschedules, and bandwidth for each threshold as percent reduction relative to DREA.

### Counting Rescheduling in DREA

DREA reschedules whenever a contingent timepoint is received or enabled. This means that between the time a contingent timepoint becomes enabled and the time it is received, DREA is continuously rescheduling. However, in practice, the simulator that we adapted is event-based, which only yields opportunity to reschedule when timepoints are received or executed. This behavior prevents DREA from rescheduling continuously between events, significantly reducing how often it reschedules. We expect that our rescheduling strategies would dramatically reduce continuous scheduling between timepoints in DREA implementations that use a time-based simulator, so their reduction in rescheduling would be even greater.

## Empirical Evaluation

The ultimate goal of our analysis is to understand the trade-off between rescheduling and success rate in our algorithms. We consider the thresholds of SI and AR, because thresholds control how selective an algorithm is about sending out new schedules, which then impacts success rate. We compare the success rates and number of reschedules of our algorithms to those of DREA and the minimally scheduling algorithms, Early First and SREA. Finally, we further examine the trade-off between rescheduling and success rate for different thresholds of AR. Through such analysis, individuals can determine which algorithm and threshold are most appropriate for their purposes.

### Impact of Thresholds

First, we compare the performance SI and AR with different thresholds against DREA. In Figure 2, we graph a percent reduction in success rate, number of reschedules, bandwidth, and simulation duration from DREA in tandem across all thresholds for a single algorithm. Our percent decrease for SI, for example, is calculated as in Equation 1.

$$Dec = 100 \cdot \frac{x_{DREA} - x_{SI}}{x_{DREA}}. \quad (1)$$

Here, $x_{DREA}$ is the average value of that metric for DREA, $x_{SI}$ is that for SI, and $Dec$ is the percent decrease in the metric. We expect a positive value, because DREA generally has higher values for all the dependent metrics. A near zero value signifies this algorithm performs similarly to DREA, and a larger value indicates a larger decrease in that metric for the algorithm. Ideally, for the new algorithms, success rate would not decrease relative to DREA; however, rescheduling, bandwidth proxy, and runtime ideally would decrease.

We make three significant observations about SI from Figure 2a. First, note that all values are positive. Since the graph depicts the percent reduction from DREA, that means our algorithms have a lower success rate as well as less rescheduling than DREA, as expected. Additionally, the metrics do not vary much across thresholds for SI; the total ranges are only $3.66\% \pm 2.52\%$ for success rate, $4.34\% \pm 0.70\%$ for number of reschedules, and $3.78\% \pm 0.76\%$ for bandwidth. Thus, our thresholds (0.1 - 0.5) have no significant impact on the functionality of the algorithm. We theorize that if we had expanded our range of thresholds to contain negative values (i.e. the new schedule is riskier than the old), then as the threshold approached -1, the performance would approach that of DREA. Such behavior occurs because a very low threshold allows the algorithm to reschedule in more cases, approaching the functionality of DREA. Likewise, in the higher range, SI may reschedule infrequently, and thus act more similarly to SREA. Finally, the percent reduction for success rate ($42.45\% \pm 0.78\%$ on average) is significantly lower than the percent reduction in number of reschedules and bandwidth ($74.26\% \pm 0.22\%$ and $67.44\% \pm 0.24\%$ on average respectively). This comparison signifies that the proportional loss in success rate is, on average, $31.81\% \pm 0.81\%$ and $24.99\% \pm 0.82\%$ smaller than the proportional loss in the respective communication metrics. Thus, SI results are promising in reducing communication to a large extent while preserving a higher success rate.

Figure 2b plots the same data across different thresholds for AR. From this plot, we see many of the same patterns present in the corresponding SI graph. All values are within error of or above zero, so AR consistently has lower success rates and communication metrics, as expected. The gaps

| AR Threshold | Reschedules | Success Rate |
|---|---|---|
| 0.1 | 1.7 | 21% |
| 0.3 | 2.8 | 22% |
| 0.5 | 3.9 | 27% |
| 0.7 | 5.8 | 30% |
| 0.9 | 6.7 | 32% |

Table 1: For each threshold of Allowable Risk (AR), we also compute the average number of reschedules and the average success rate across all runs over all inputs.

between the success rate with number of reschedule and bandwidth reductions are smaller ($16.84\% \pm 0.84\%$ and $12.14\% \pm 0.84\%$ on average respectively) so AR is less effective at lowering rescheduling than SI. However, AR does a better job of maintaining success rate. Its success rate is within error of DREA's at threshold values of .7 and .9, where success rate is reduced by $1.14\% \pm 3.12\%$ and $-2.78\% \pm 3.22\%$ compared to DREA, respectively.

The main difference is that the performance of AR varies with respect to threshold. As the threshold decreases, the success rate and rescheduling reductions all decrease at approximately the same rate (on average, $9.67\% \pm 1.17\%$ and $12.75\% \pm 0.46\%$ per 0.2 units of threshold respectively). We expect this result, because at a threshold of 0, AR will always reschedule, and thus act like DREA, while at a threshold of 1, it will never reschedule, and thus act like SREA. This graph explores thresholds in between and depicts a reasonable downward trend. AR also decreases communication more than success rate. In addition, varying the parameter depicts a clear trade-off between low thresholds (following low communication) with low success rate. Therefore, AR is tunable across the range 0.1 to 0.9.

In Figure 2, we also map the percent reduction for runtime. For AR, the runtime depicts the same general downward trend across threshold as the other metrics. This downward trend is expected, because AR refrains from computing new schedules when it decides against sending one. SI does not exhibit this same property; it always calculates a new schedule, then decides whether to send it out.

### Trade-offs Within Allowable Risk

We also attempt to more clearly understand the magnitude trade-offs between number of reschedules and success rate in AR. In Table 1, we calculate AR's average success rate and number of reschedules for a given threshold. Note, this table represents the same experiment as in Figure 2b, but this time we present the absolute (vs. relative) success rate and number of reschedules. We observe that average success rate tends to increase as average number of reschedules increases. We would expect this trade-off, because the higher the number of reschedules, the more information used in rescheduling, the better the performance. Note this pattern is not evident for SI.

Across our analyses, we explore a number of ways to represent the trade-offs between success rate and communication. Our algorithms successfully explore the space between SREA and DREA in terms of amount of communication and success rate. In addition, we find the AR threshold corresponds well with amount of communication, and thus provides a good basis for directly analyzing the trade-off between communication and success rate.

### Impact of Problem Features

Beyond evaluating threshold effects, we compare our algorithms with DREA, SREA, and Early First across different classes of inputs. Figure 3 show two examples of simple average metric comparison across different algorithms. For all algorithms, we graph the success rate, number of reschedules, and bandwidth as functions of degree of synchrony, interagent constraint density, and standard deviation. This arrangement yields a unique graph for each pairing of a dependent and independent variable (ie, success rate and degree of synchrony). With these graphs, we can directly compare the performance of all parameters for each metric across all input features.

In this paper, we only show success rate and number of reschedules for degree of synchrony in Figure 3. We selected these graphs to show clear trends from several input values. However, the same trends are also present for interagent constraint density and standard deviation.

Figure 3a shows that the success rate of all of our algorithms is consistently between that of DREA and SREA, as expected. Our algorithms all are modifications of DREA that schedule less, and therefore don't utilize as much information. However, they all still reschedule more than SREA. It is also noteworthy that SI and AR outperform Early First, while only CT seems to track Early First, even though Early First never communicates. Therefore, CT failed to preserve a large enough success rate to make rescheduling worthwhile in comparison.

In addition, we note that as degree of synchrony increases, success rate increases. Such a correlation makes sense, because the higher the value, the larger the possible acceptable time range for constraints between agents. The constraints are simply less strict, and thus easier to satisfy.

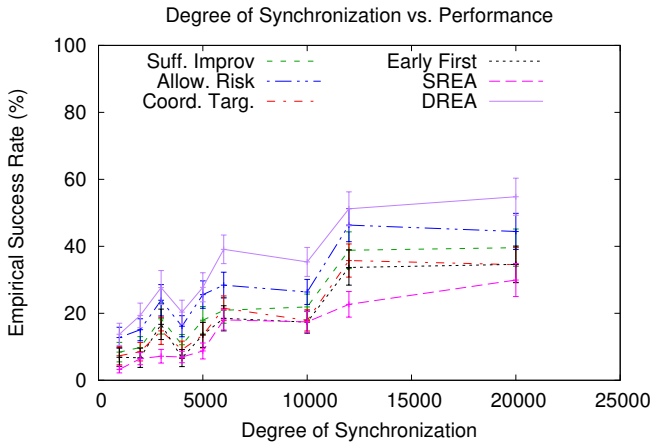
Similarly, Figure 3b shows that the number of reschedules of all of our algorithms is consistently between DREA and SREA. SI reschedules at a rate close to SREA, which is optimal. CT and AR both consistently outperform DREA, but are not as good as SI in this sense. Lastly, there is no trend in number of reschedules as degree of synchrony increases.

Figure 3, as well as all graphs of this class, additionally express that the aforementioned results hold across different input feature values. The input features are designed to represent important aspects of scenarios, implying our conclusions hold for many different types of scenarios. Our results are thus more generalizable.

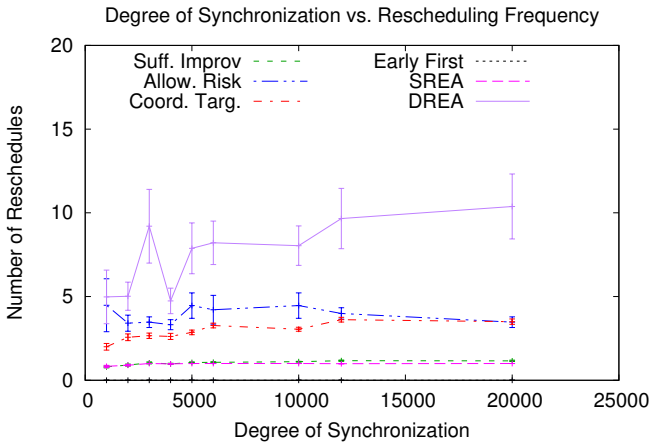
## Discussion

In this paper, we augment Lund et al. (2017)'s DREA to maintain its high success rate while simultaneously reducing its high number of reschedules. To this end, we propose three new algorithms: Sufficient Improvement (SI), Allowable Risk (AR), and Coordination Targeting (CT).

Our exploration of these algorithms and their parameters shows a clear trade-off between rescheduling and success. We are therefore unable to conclude which of our algorithms is best, because it depends on the relative importance of rescheduling and success for a given application.




(a) Success rate across degree of synchrony




(b) Number of reschedules across degree of synchrony

Figure 3: Results of all our algorithms across different degrees of synchronization. The thresholds were selected to be intermediate values in the tested range, 0.3 for Sufficient Improvement (SI) and 0.5 for Allowable Risk (AR).

SI and AR show promise, where CT was unable to outperform Early First. Results for SI and AR are consistent for simulations that were run and evaluated on PSTNs varying across a number of identified features. Our analysis is thus generalizable to many different kinds of input scenarios.

SI has an impressive gap between its percent reduction of success rate and communication with respect to DREA. This difference means that the loss in algorithm success for the amount it reschedules is relatively low, resulting in a less costly trade-off between success rate and communication. Unfortunately, changing the SI threshold does not appear to affect success rate or communication. Therefore, threshold tuning within our explored range has a negligible effect on the performance of the algorithm. Future work might consider exploring other values of the SI threshold outside our range, particularly negative values.

While AR does not decrease communication as much as SI, for certain thresholds it reaches much higher success rates. Most importantly, AR exhibits a clear trade-off: as threshold increases, amount of communication decreases and success rate increases. Thus, AR is more tunable to be appropriate for the situation in which it is used. In scenarios where communication is especially costly, like in our UAV example, lower thresholds are preferable. However, if communication is only slightly limited, the AR threshold can be increased to capture a larger success rate. Thus, AR is more adjustable for various situations than SI.

As we can see, there is no clear winner between SI and AR; each algorithm may be preferable in different situations. Moreover, in scenarios where communication cost is negligible, DREA may be more suitable, and in scenarios where communication cost is extreme, Early First might be best. In any case, our contributions have successfully explored the trade-offs between success rate and communication in the region between DREA and static algorithms.

Future research could explore combining these algorithms in an attempt to reduce scheduling further. SI might, for example, be layered on top of AR. Such an algorithm might vary its rescheduling frequency based on the risk present in the system (like AR), and then only sends out new schedules if they constitute a significant improvement over the current one. SI itself might attain higher robustness (probably accompanied by a higher rescheduling rate) by rescheduling when new schedules constitute a significant *change* in predicted success rate, rather than a significant *improvement* in predicted success rate.

Another possible direction for future work is to test these algorithms in other kinds of simulations. For instance, one could run tests in a time-based simulation, as opposed to an event-based simulation. It would be interesting to see how DREA, or an adapted version of our algorithms, would perform in this context. Finally, physics-based or real-world simulations would yield results more directly relatable to practical situations. This would create a stronger basis for justifying the value of SI and AR, and defining the trade-off between success rate and communication.

## Acknowledgements

We would like to thank the Clinic organizers, Professor Zachary Dodds and DruAnn Thomas. We would also like to thank Jordan Abrahams, Hamzah Khan, Kyle Lund, and Brenner Ryan for lending us their code and expertise. This research was supported by the NASA Advanced Exploration Systems (AES) program.

## References

Brooks, J.; Reed, E.; Gruver, A.; and Boerkoel, J. C. 2015. Robustness in probabilistic temporal planning. In *Proc. of the 29$^{th}$ National Conference on Artificial Intelligence (AAAI-15)*, 3239–3246.

Cesare, K.; Skeele, R.; Yoo, S.; Zhang, Y.; and Hollinger, G. 2015. Multi-UAV exploration with limited communication and battery. In *Proc. of the IEEE Conference on Robotics and Automation (ICRA)*, 2230 – 2235.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. In *Knowledge Representation*, volume 49, 61–95.

Fang, C.; Yu, P.; and Williams, B. C. 2014. Chance-constrained probabilistic simple temporal problems. In

*Proc. of the 28$^{th}$ National Conference on Artificial Intelligence (AAAI-16)*, 2264–2270.

Li, B.; Jiang, Y.; Sun, J.; Cai, L.; and Wen, C.-Y. 2016. Development and testing of a two-UAV communication relay system. *Sensors* 16(10).

Lund, K.; Dietrich, S.; Chow, S.; and Boerkoel, J. 2017. Robust execution of probabilistic temporal plans. In *Proc. of the 31$^{st}$ National Conference on Artificial Intelligence (AAAI-17)*, 3597–3604.

Quach, C.; Bole, B.; Hogge, E.; Vazquez, S.; Daigle, M.; Celaya, J.; Weber, A.; and Goebel, K. 2013. Battery charge depletion prediction on an electric aircraft. In *Proc. of the Annual Conference of the Prognostics and Health Management Society (PHM 2013))*.

Tsamardinos, I. 2002. A probabilistic approach to robust execution of temporal plans with uncertainty. In *Methods and Applications of Artificial Intelligence*. Springer. 97–108.

Vidal, T., and Ghallab, M. 1996. Dealing with uncertain durations in temporal constraint networks dedicated to planning. In *Proc. of European Conference on Artificial Intelligence (ECAI-96)*, 48–54.

Wu, F.; Zilberstein, S.; and Chen, X. 2011. Online planning for multi-agent systems with bounded communication. *Artificial Intelligence* (175):487 – 511.

Zhang, J.; Zheng, Y.; and Zheng, R. 2017. Spectrum and energy efficiency maximization in UAV-enabled mobile relaying. In *Proc. of IEEE International Conference on Communications (ICC))*.

# 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.

**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-

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

[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`.

```
(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).

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 retrieve 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 by 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.

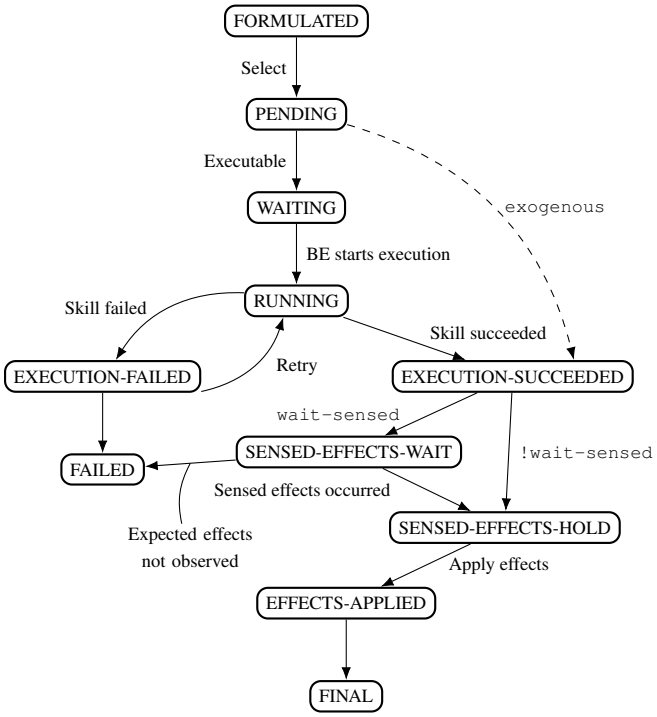



Figure 1: The possible states of a `plan-action` and the transitions between those states.

### 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`.

# Planning and Execution for Front Delineation and Tracking with Multiple Underwater Vehicles

**Andrew Branch[1], Mar M. Flexas[2], Brian Claus[3], Andrew F. Thompson[2], Evan B. Clark[1], Yanwu Zhang[4], James C. Kinsey[3], Steve Chien[1], David M. Fratantoni[5], Brett Hobson[4], Brian Kieft[4], Francisco P. Chavez[4]**

[1]Jet Propulsion Laboratory, California Institute of Technology
[2]California Institute of Technology
[3]Woods Hole Oceanographic Institution
[4]Monterey Bay Aquarium Research Institute
[5]Remote Sensing Solutions
Correspondence Author: andrew.branch@jpl.nasa.gov

**Abstract**

This work describes a planning architecture for a heterogeneous fleet of marine assets as well as a method for detecting and tracking ocean fronts using multiple autonomous underwater vehicles. Multiple vehicles — equally-spaced along the expected frontal boundary — complete near parallel transects orthogonal to the front. Lateral gradients are used to determine the location of the front crossing from each individual vehicle transect by detecting a change in the observed water property. Adaptive control of the vehicles ensure they remain perpendicular to the estimated frontal boundary as it evolves over time. This method was demonstrated in several experiment periods totaling weeks, in and around Monterey Bay, California in May and June of 2017. We discuss the challenges associated with the implementation of the planning system. We show the capability of this method for repeated sampling across a dynamic two-dimensional ocean front using a fleet of three types of platforms: short-range Iver AUVs, Tethys-Class Long-Range AUVs, and Seagliders. This method extends to tracking gradients of different properties using a variety of vehicles.

## Introduction

Space-based remote sensing can provide extensive information about ocean dynamics. However, remote sensing information is generally limited to measuring the ocean surface. To probe the ocean interior efficiently requires marine vehicles such as autonomous underwater vehicles (AUVs), gliders, profiling buoys, surface vehicles, and ships sampling in situ. Unfortunately, building, deploying and operating these in situ marine robotic explorers is expensive. As a result, any actual study involves a limited number of marine vehicles, especially when compared to the vast expanse of the ocean. Determining where to deploy and operate marine assets is a challenging problem given the 4D spatiotemporal variations in oceanographic phenomena.

The use of autonomous marine vehicles will increase as the size of ocean observing systems expand in order to study the impact of the oceans on Earth's climate and ecosystems. The day-to-day operations of these systems will become increasingly difficult if human intervention is required. In order to enable large observing systems to operate, techniques for autonomous control of assets based on science goals and data sources such as in situ measurements, remote-sensing, and model-derived data need to be developed. Such observing systems will incorporate a wide variety of vehicles with differing capabilities. Planning and execution systems that leverage existing infrastructure help to reduce the cost associated with the development and maintenance of an observing system as well as maintain flexibility with regards to the planning approach and vehicle availability. The Keck Institute for Space Studies (KISS) Satellites to Seafloor project works towards this goal of fully autonomous sampling [Thompson et al., 2017]. Previous ocean observing systems have relied on substantial human intervention or non-adaptive sampling strategies, including the Autonomous Ocean Sampling Networks (AOSN) [Curtin and Bellingham, 2009; Curtin et al., 1993; Haley et al., 2009; Leonard et al., 2007; Ramp et al., 2009] and the Adaptive Sampling and Prediction (ASAP) [Leonard et al., 2010] projects.

Our project targets automatic generation of coordinated mission plans for teams of assets to follow science derived observation policies (e.g. the use of multiple vehicles to perform transects orthogonal to an ocean front). This paper describes a planning and execution system for a heterogeneous fleet of marine assets. To highlight this system, an approach was developed using multiple vehicles to make a linear estimation of an ocean front's geometry and to continuously direct a team of marine robotic vehicles to perform orthogonal transects with the midpoint of the transect roughly centered on the target front. We describe both the general approach to front-crossing detection, front-geometry estimation, and multi-asset control, the architecture of the planning and execution system for a deployment using three types of vehicles: short-range Iver Autonomous Underwater Vehicles, Long-Range Tethys Autonomous Underwater vehicles, and long-range Seaglider buoyancy driven gliders in Monterey Bay in late spring 2017, the results from the deployment, and the challenges associ-

ated with this system. This deployment was the result of a team effort between the KISS project members and the MBARI Spring 2017 CANON participants [Monterey Bay Aquarium Research Institute, 2017]. The method and systems presented here represent significant steps towards the fully-autonomous adaptive sampling framework as envisioned in Thompson et al. [2017].

## Front-Crossing Detection

### Lateral Gradient Front-Crossing Detection

The KISS team developed an algorithm to identify a subsurface oceanic front based on lateral gradients of a given hydrographic property. This could be temperature, buoyancy or density (if salinity data is available), or any available biogeochemical property such as dissolved oxygen or chlorophyll.

When in situ data is received in near real time, the algorithm grids the data, smooths it by applying a simple linear weighted average of immediate neighboring measured data points, and calculates the lateral gradients (Figure 1). The algorithm uses temporal gradients, and assumes that time can be linearly related to distance. The algorithm then calculates the lateral gradients along the transect within the layer of interest (defined beforehand by the user) as well as the mean value, and the standard deviation. The user also defines beforehand the number of standard deviations used to declare a front-crossing detection. All points above this threshold are considered potential front crossings (Figure 2). To qualify for a frontal crossing, it is required that the threshold is crossed twice (once entering and once leaving the high gradient region). The width of the front is used to choose the front crossing of interest if more than one is present. The front location, width, and time of crossing is then output for later use in vehicle tasking. An example is shown in Figure 1 and Figure 2. Time, as apposed to distance, is plotted on the x-axis as that is what the algorithm uses. Using real time data from May 4, 2017 (Figure 1d) the algorithm detects five narrow subsurface fronts from 10 to 15 m deep (Figure 2a), and selects the widest front (Figure 2d).

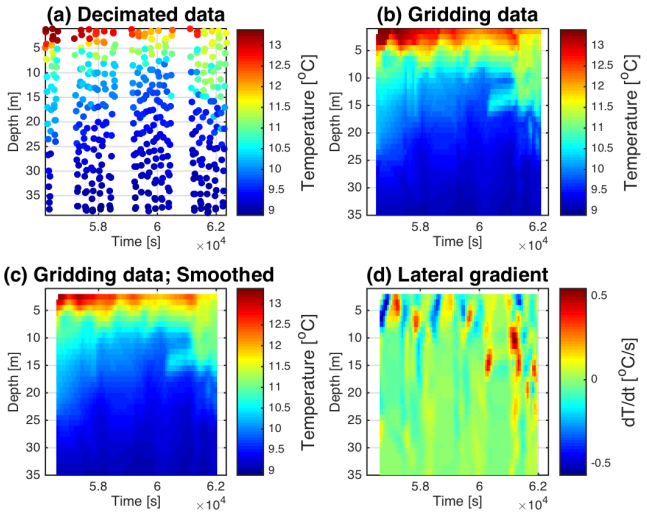



Figure 1: Lateral gradient front-crossing detector. For this example we use data obtained on May 4, 2017 from Iver 136 (segment 000). Real-time in situ temperature data (shown in scatter plot in panel a) is gridded (panel b) and smoothed (panel c). Then, lateral gradients are calculated (panel d). When used in real time, the algorithm uses temporal gradients, and assumes that time can be linearly related to distance.

## Autonomous Control of Underwater Vehicles for Front Tracking

A technique was developed to control a group of vehicles to repeatedly sample across a dynamic ocean front as it evolves over time. The planner must be able to modify the vehicle transects in order to adapt to the changing ocean conditions. The control algorithm is outlined in Algorithm 1 and shown in Figure 3. The statements in which the planning system interacts with the execution system (i.e. the vehicles) are highlighted. When first deployed, an initial estimated front location and orientation is manually provided based on available data from other assets. The vehicles are equally spaced along this estimated front. Each vehicle is commanded on an initial transect orthogonal to the estimated front. When the vehicle surfaces to plan, Algorithm 1 is executed. The vehicle location and the scientific data from the current transect are retrieved from the execution system as *vehicle_location* and *transect_data* respectively. The vehicles location along the transect is calculated as $location_p$ by projecting the vehicles current location onto the commanded transect. If the vehicle has traveled a minimum distance along the commanded transect, specified by $transect\_dist_{min}$, then the front-crossing detection algorithm is run on the data from this transect. The resulting front-crossing is defined as $new\_front\_crossing$. If the vehicle is a specified distance past this new front detection, then the front is re-estimated using linear regression on front detections from all vehicles, otherwise the transect is continued. When re-estimating, only certain front detections from each vehicle are considered, specified by $valid\_front\_detections$. We used two methods when selecting the subset of detections used in the linear regression: a time based approach where detections from the last $N$ hours were considered and a latest detection approach where only the last detection from each vehicle was considered. These two approaches are defined in the procedure $get\_estimation\_crossings$. The new $transect_p$ is calculated such that it is orthogonal to $estimated\_front$. The vehicle is then commanded on this new transect. In order to prevent the vehicle from leaving the study area, $transect\_dist_{max}$ is defined. If a transect has reached this length the front is re-estimated, a transect orthogonal to this is defined,

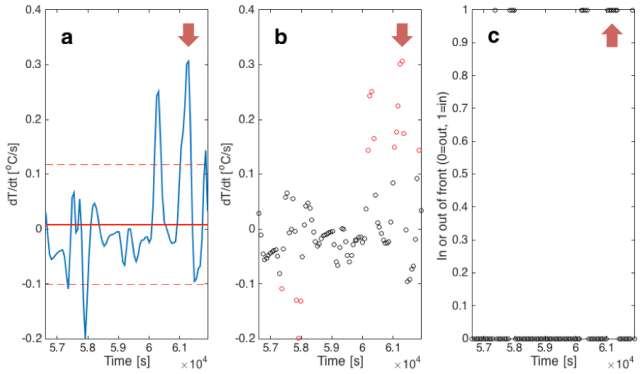



Figure 2: (Continues from Figure 1) The algorithm calculates the mean value of the lateral gradients over the layer of interest. In this example, we use data from 10m to 15m. The algorithm calculates the mean value (bold red line in panel a) and the n-standard deviation (in this case, n=1.2; red broken lines in panel a). All points above the n-value standard deviation are considered potential fronts (red circles in panel b). A boolean is used to isolate the front crossings (panel c). The width of the front is used to choose the front crossing when more than one front is present. The crossing chosen by the algorithm is marked with a red arrow.

and the vehicle is commanded on this new transect.

## Pilot Experiment

### Experiment Site

The pilot experiment occurred in Monterey Bay, California (36.80°N, 121.90°W) from May to June 2017. The circulation in Monterey Bay is characterized by a persistent coastal upwelling, in response to prevalent northerly winds, which generates highly-productive cold coastal regions [Hickey, 1979; Lynn and Simpson, 1987]. In May 2017, an intensive upwelling plume spread southeastward across the mouth of Monterey Bay. A fleet of AUVs were deployed to detect and track the fronts between the upwelling plume and the stratified inner bay water. Over the shelf, KISS IVERs were set to detect lateral gradients of temperature from 10m to 15m. Over the slope, temperature in the upwelling water column was remarkably homogeneous in the vertical dimension. Over the slope, MBARI LRAUVs were also set to detect lateral gradients of temperatures from 10m to 15m.

Glider retasking took place in June 2017, offshore Point Sur, where the California Undercurrent (CU) becomes unstable [Molemaker, McWilliams, and Dewar, 2015]. Looking for the surface signature of the CU, one Seaglider was set to detect lateral gradients of temperature from 5m to 15m. The operations regions for each vehicle are shown in Figure 4.

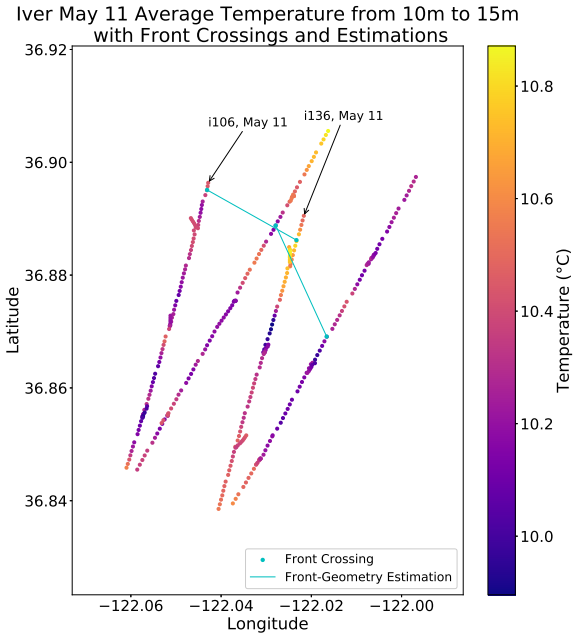



Figure 3: Iver transects on May 11 with temperature averaged from 10 meters to 15 meters plotted. Front crossings are shown as blue dots and estimated fronts are shown as blue lines. Each vehicle starting location is labeled with the vehicle name and the date. The second transect for each vehicle is orthogonal to the estimated front from the front crossings on the first transect.

### Instruments

This work was demonstrated across three types of underwater vehicles: the OceanServer Iver AUV, the Kongsberg Underwater Technology, Inc. Seaglider and the MBARI Tethys-class LRAUV (shown in Figure 5). The method is extensible to other platforms and indeed other domains where the vehicles are able to at least intermittently transmit collected data and receive new instructions mid-deployment.

**Iver AUVs** The highest speed observing platforms used for this field experiment consisted of two Iver2 (Ocean Server Technology Inc.) autonomous underwater vehicles (AUVs) [Crowell, 2006]. Both of the vehicles were equipped with a hull-mounted Neil Brown conductivity/temperature sensor (Ocean Sensors Inc.) which served as the primary scientific payload for this work. Additionally, one of these vehicles, Iver-106, was an Ecomapper variant equipped with a SonTek Doppler velocity log (DVL), an Ocean-Server compass for attitude estimation, a WHOI micro-modem 2 and a depth sensor. The other Iver2 vehicle, Iver-136, was similarly equipped with the WHOI micro-modem 2, compass and depth sensor as well as a dual upward, downward facing 600 kHz RDI phased array DVL, a Microstrain 3DM-

**Algorithm 1** Linear Front Delineation and Tracking

**procedure** VEHICLE_RETASKING ▷ Run this procedure when a vehicle surfaces to plan
  $vehicle\_location \leftarrow$ Get vehicle location
  $transect\_data \leftarrow$ Get vehicle data
  $location_p \leftarrow project\,(transect, vehicle\_location)$
  **if** $dist\left(transect\_start, location_p\right) >= transect\_dist_{min}$ **then**
    $new\_crossing \leftarrow detect\_crossings\,(transect\_data)$
    **if** $new\_crossing$ was detected **then**
      $crossings \leftarrow crossings \bigcup \{new\_crossing\}$
      $valid\_crossings \leftarrow get\_estimation\_crossings(crossings)$
      $estimated\_front \leftarrow linear\_regression\,(valid\_crossings)$
      $location_f \leftarrow project\,(transect, new\_front\_crossing)$
      **if** $dist\left(location_p, location_f\right) > \varepsilon_{past\_front}$ $km$ **then**
        Calculate $transect_p$ **s.t.** $transect_p \perp estimated\_front$
        Command vehicle on $transect_p$
      **else**
        Continue on current transect
  **else if** $dist\left(transect\_start, location_p\right) <= transect\_dist_{max}$ **then**
    $valid\_crossings \leftarrow get\_estimation\_crossings(crossings)$
    $estimated\_front \leftarrow linear\_regression\,(valid\_crossings)$
    Calculate $transect_p$ **s.t.** $transect_p \perp estimated\_front$
    Command vehicle on $transect_p$

**procedure** GET_ESTIMATION_CROSSINGS($crossings$) ▷ First of two options for this procedure
  **return** Latest front crossing for each vehicle.

**procedure** GET_ESTIMATION_CROSSINGS($crossings$) ▷ Second of two options for this procedure
  **return** $\{crossing \in crossings \mid crossing.time > current\_time - \varepsilon_{time}\}$

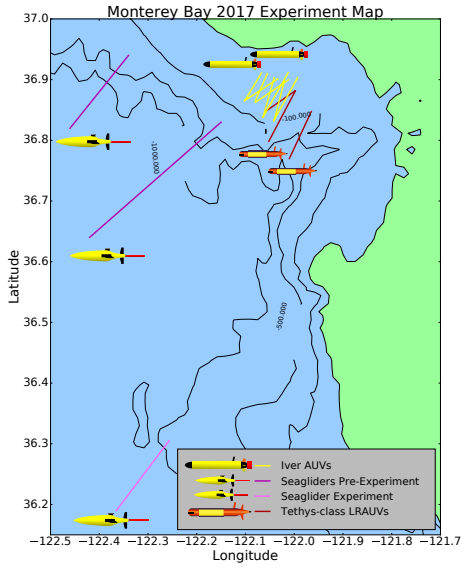



Figure 4: Map of the 2017 pilot experiment region near Monterey Bay, California. The operation regions of the Iver AUVs, Seagliders, and Tethys-class LRAUVs are shown.

GX3-25 and an APS-1540 fluxgate magnetometer. The Iver2 AUVs have an approximate maximum horizontal velocity of 2 m $s^{-1}$ and were operated at a speed of 1.5 m $s^{-1}$ for these trials. These vehicles are shown on board the R/V *Shana Rae* in Figure 5 during operations in August 2016.

**Long-Range AUVs** Also used in this experiment were two Tethys-Class Long-Range AUVs (Monterey Bay Aquarium Research Institute) [Bellingham et al., 2010; Hobson et al., 2012] (Figure 5). Each vehicle was equipped with a Neil Brown conductivity, temperature, depth (CTD) sensor and a Sea-Bird ECO fluorometer and backscattering sensor. The LRAUVs have an approximate maximum horizontal velocity of 1 m $s^{-1}$ and an endurance of 1,000+ km. The vehicle is capable of sampling to a maximum depth of 200 m in a saw–tooth pattern (i.e. yo-yo). An iridium modem is used for sending commands to the vehicle as well as downloading a subset of the data to the shore. When cellular signal is available, a cellular modem is used to send the full dataset.

**Underwater Gliders** We used two Seagliders (Kongsberg Underwater Technology, Inc.) [Eriksen et al., 2001] equipped with Seabird SBE3 temperature sensor and SBE4 conductivity sensor, pressure sensor, and Aanderaa 4330F oxygen optode (Figure 5). Sampling occurred approximately every 5 s (0.5 m vertical resolution at typical vertical speeds of 0.1 m $s^{-1}$). The gliders use a buoyancy engine for propulsion, having an approximate horizontal velocity of 0.25 m $s^{-1}$ and endurance up to 4,600 km. For this experiment, we were tasking the gliders to maximum depths of 600 m (although they are capable of profiling to a maximum depth of 1000 m) in a saw–tooth pattern.

## System Architecture

Existing systems are leveraged to deploy a general planning method to a variety of vehicles in a short time frame. The system architecture for all vehicles used in this experiment is shown in Figure 6. The Seaglider and LRAUV both operate remotely using the Iridium network. On each surfacing, GPS, engineering, and scientific data transmits to a shore-based control workstation. This workstation can then issue commands to the vehicle. For this experiment, the planning software ran on a separate shore-based workstation capable of communicating with the workstation controlling the vehicles. In this way it was possible for the planner to receive all the necessary data and send commands to the vehicles in near real-time.

While the Seaglider and LRAUV are nominally able to transmit data and receive new instructions during operations, the Iver AUVs required some modifications to enable these behaviors. Four communication modalities are available to the Iver: Iridium short burst data (SBD), Wi-Fi, 900 MHz RF, and acoustic modem. Scientific data such as position, conductivity, temperature, and timestamps can be received and new commands can be sent over any of these four available communication links. Possible commands include stopping a mission,

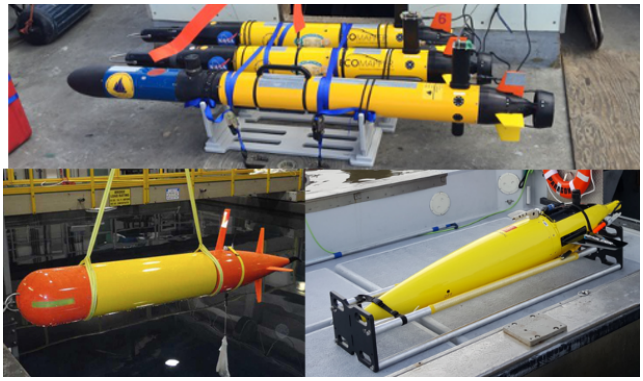


Figure 5: Top: OceanServer Technology, Inc. Iver2 AUVs onboard the R/V *Shana Rae*, Bottom Left: Monterey Bay Aquarium Research Institute's Tethys-Class Long-Range AUV., Bottom Right: Kongsberg Underwater Technology, Inc. Seaglider onboard the R/V *Paragon*

starting a mission already loaded on the vehicle, parking the vehicle and inserting segments of waypoints into the already running mission. Initially, it was planned to use the segment insertion to facilitate the retasking of the vehicles. While these commands were successfully received and interpreted by the vehicle, some unexplained behaviors while using this command precluded its ongoing use. As a temporary work around for the 2017 field trials in Monterey we used the outputs of the planning software to manually program a new mission which was then loaded onto the AUV over the RF link. Due to the short range of the Iver AUVs, a surface vessel remains deployed near the vehicles at all times. This surface vessel also houses the control and planning workstations for the vehicle. The Iridium link was active during this experiment, but was not used for planning purposes.

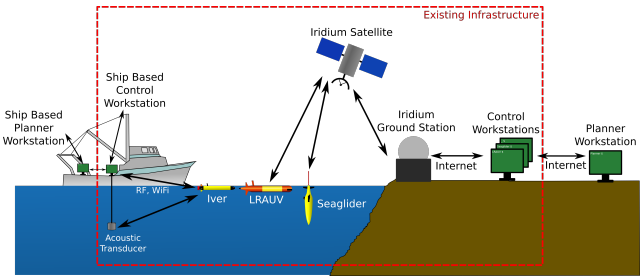



Figure 6: System diagram outlining the communication pathways from the vehicles to the controlling workstations and the planning workstations. The existing infrastructure is outlined with a red dotted line.

## Results

An abridged version of the results are presented here. The full results can be found in Branch et al. [2018].

### Iver AUV Results

Two Iver AUVs were operated on three days, 4 May, 9 May, and 11 May 2017. They are limited to single day deployments due to the short range of the vehicles. Some operational constraints required modifications to the outlined front tracking control method. The range limitation associated with acoustic and RF communication and the desire to have the ability for quick vehicle recovery required the two Iver AUVs to remain in close proximity to each other. The front tracking algorithm as presented does not guarantee any vehicle synchronization with regards to position. To solve this issue, the vehicles pause at any point in which a new transect could start and waits for every other vehicle to reach their respective decision points. Once all vehicles have paused, the front-crossing detection algorithms are executed for each vehicle. If at least one vehicle has detected a front crossing, a new linear front estimation is generated and all vehicles are commanded orthogonal to it. If no front crossings are detected then all vehicles continue on the current transect.

In this experiment the minimum transect distance was set at 3 km past the current estimated front. The minimum distance required for a vehicle to go past the front-crossing detection on a given transect was set to 0 km, this results in the vehicle turning around at the first decision point after a front crossing is detected. The first decision point can be significantly past the detected front crossing due the minimum transect length. Ideally this would be set to a longer distance to insure that the vehicle has crossed the entire front before calculating a new transect, however due to software constraints for this phase of the deployment this was not possible. Front-geometry estimation was performed with the latest front crossing from each vehicle. The lateral gradient front-crossing detection algorithm was used with the Iver AUVs. Figure 7 shows the results of the Iver experiment on 9 and 11 May, 2017. Two transects were completed per vehicle per day. The starting locations for each vehicle on each day are labeled. Temperature averaged from 10 meters to 15 meters is plotted. All front crossing and front-geometry estimations used during the deployment are shown as blue dots and blue lines respectively. A number of different depth intervals for front-crossing detection were used during the deployment in order to examine the sensitivity of the algorithm. For reference, the front crossings and front-geometry estimations for 10 meter to 15 meter depth range are also plotted in green.

### LRAUV Results

The LRAUV experiment took place on 07 May, 2017. Two vehicles, Opah and Tethys, were under the control of the planner and utilized the lateral gradient front-crossing detection method. The minimum transect distance was set at 4.5 km past the current estimated front. The minimum distance required for a vehicle to go past the front-crossing detection on a given transect was set to 0 km, this results in the vehicle turning around at

*Underwater Glider Results*

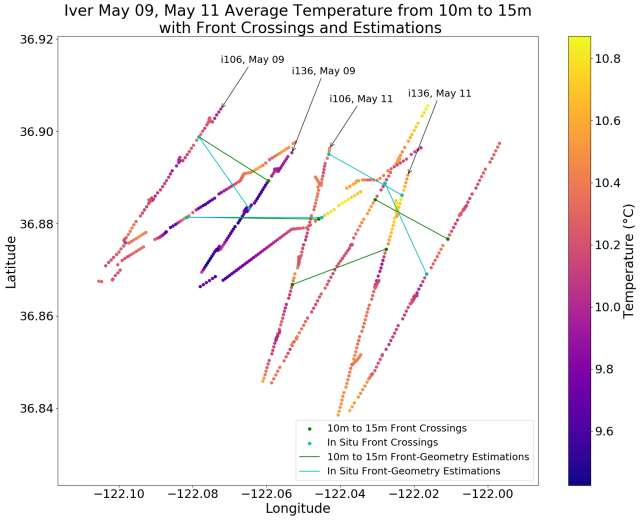



Figure 7: Map view of the temperature averaged from 10 to 15 meters for the Iver transects on 09 and 11 May, 2017. Front crossings and front-geometry estimations used during the experiment are indicated with a blue dot and blue line respectively. Front crossings and front-geometry estimations using data from 10 meters to 15 meters during the experiment are indicated with a green dot and green line respectively. The start location for each vehicle for each day is labeled.

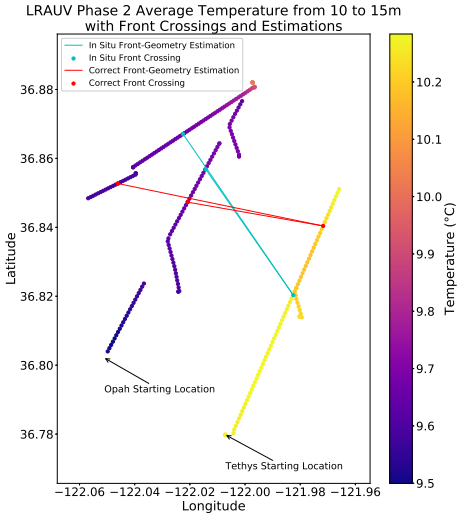



Figure 8: Map view of the temperature averaged from 10 to 15 meters for the LRAUV Phase 2 experiment. The front crossings and front-geometry estimations used in the deployment are plotted as blue dots and blue lines respectively. The correctly calculated front crossing and front-geometry estimations are plotted as red dots and red lines respectively.

the first decision point after a front crossing is detected. Once again, the minimum distance past a front crossing would ideally be larger. Front-geometry estimation was performed with the latest front crossings from each vehicle. Figure 8 shows the results from the phase 2 of the LRAUV experiment. The temperature averaged from 10m to 15m, the interval used for front-crossing detection, is plotted. The algorithm during this period of the deployment ran incorrectly, resulting in erroneous front crossings. The algorithm was re-run correctly in post-processing. Both the front crossings used during the deployment and the correct front crossings are plotted in Figure 8. Opah was able to complete two transect while Tethys only completed one transect due to hardware issues.

## Underwater Glider Results

The underwater glider operated off the coast of Point Sur, California from 7 June to 21 June, 2017. From 7 June to 15 June the glider was in a region of strong surface currents, preventing any significant forward movement. The glider transect was relocated and successfully operated from 15 June to 21 June, 2017. During the glider portion of the experiment, only one vehicle was available. Using the method presented here, it is not possible to estimate the orientation of a linear front with a single vehicle. As such, a fixed transect orientation is used in this experiment. The minimum transect distance and the minimum distance to travel past a front were set to a fixed 5 km, independent of the current location of the estimated front. In normal operation these distances would be increased. Due to the short time frame of the experiment these were reduced in order to complete more transects.

A map view of the 6 glider transects plotting the averaged temperature over 10 meters to 15 meters, the interval used for front-crossing detection, can be seen in Figure 9. The front crossings and front estimations are marked with a blue dot and a blue line respectively. The 16 km maximum extent transect is shown in black.

## Planning and Execution Challenges

**Communication Paradigms** The LRAUV and the Seaglider both utilize the Iridium network to enable the off-board planning system to control the vehicle. A centralized off-board planner simplifies vehicle coordination and allows for the use of a variety of vehicles while avoiding unique on-board implementations. This comes at the cost of reduced real time capabilities as vehicles are unable to transmit data and receive new plans during a dive. The default schedule of surfacing activities of the LRAUV and Seaglider also impacts the real time capabilities of the system. Immediately after the data is received from the vehicle, the Iridium connection is closed and the vehicle dives. This induces a one dive delay when using the data from the vehicle for planning purposes. The surfacing schedule can be modified in order to remove this, at the cost of increased

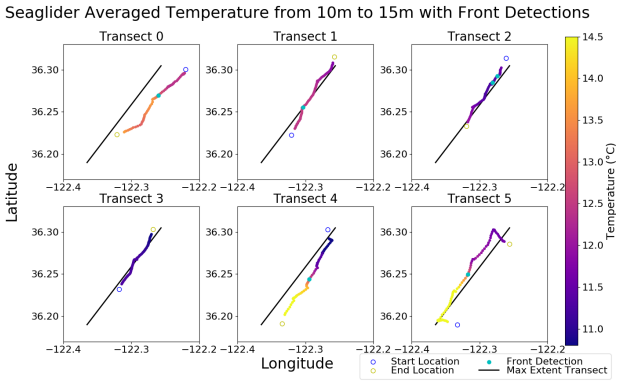



Figure 9: Map view plot of the temperature averaged from 10 to 15 meters for all the glider transects from 15 to 21 June, 2017 off Point Sur, California. Front crossings are indicated with blue dots.

surfacing times.

The Ivers use a similar off-board planning system, however it is ship-based as apposed to shore-based. The range limitations associated with acoustic, RF, and Wi-Fi communication impose additional constraints. It is ideal for the vehicles to be in close proximity so the ship remains in contact with all vehicles simultaneously. Our specific planning approach was modified in order to accommodate this. The real time capabilities could also be improved by utilizing the acoustic communication channel for scientific data and vehicle commanding. Note that the Iridium network is a possible communication modality for the Iver, but was not used during this deployment so the nearby ship could maintain full control of the vehicles.

**Data Decimation** The bandwidth of the communication channels available is not always large enough to support the transfer of the complete dataset acquired by the vehicle. When this is the case, a subset of the data must be selected for transmission and use by the off-board planner. The Seaglider is capable of sending the full dataset at each surfacing, however the other two vehicles are not. The Iver AUVs selects data at a fixed temporal resolution. The Tethys-Class LRAUV selects data based on the change from the previously transmitted data point. If a given data point differs by a specified amount from the previously selected data point then the given data point is also selected for transmission. During the experiment, we recognized that the decimated dataset from the LRAUV contained large gaps, resulting in suboptimal gridded data. An appropriate data decimation scheme needs to be employed for a given planning method.

**Vehicle Safety** Vehicle safety concerns must be addressed when implementing a planning system. A concern present with all vehicles is contact with the seafloor. The three vehicles used in the experiment have the capability of autonomously avoiding the seafloor using a sonar based device. However, to increase vehicle endurance, these devices were disabled on the LRAUVs and Seagliders. Instead, an additional layer was added to the planner in order to avoid seafloor collisions. The Seaglider dive depth was altered based on the bathymetry along the expected dive path, while the LRAUV's planned transects were modified to avoid areas with bathymetery less than a specified depth.

A related concern is the lateral position of the vehicles. Each vehicle must remain in the target region. Due to the short experiment periods and limited transect length for the LRAUVs and Ivers, this was not a concern. The Seaglider deployment used boundaries to limit the transect and prevent the vehicle from moving onto the continental shelf. It is also desirable for Ivers to remain in close proximity so the surface ship with the control workstations can be in range of all vehicles simultaneously and quick recoveries are possible. The planning approach was modified to satisfy this constraint.

## Related Work

Adaptive sampling and control of multiple autonomous underwater vehicles has been extensively studied, including foundational work with the Autonomous Ocean Sampling Network [Curtin and Bellingham, 2009; Curtin et al., 1993; Haley et al., 2009; Leonard et al., 2007; Ramp et al., 2009]. Much of this work focuses on spatially adapting the control strategy in order to optimally sample a fixed region. The Adaptive Sampling and Prediction project [Leonard et al., 2010] used adaptive control in order to coordinate 6 gliders to fly in loops at fixed spacing. Our method instead performs repeated focused sampling across a single front as it evolves over time.

Other work focused on control strategies that adapt to the current conditions, however not using multi-vehicle coordination. Troesch et al. [2016] uses an ocean model in order to improve the station keeping ability of vertically profiling floats. Eriksen et al. [2001] describes the capabilities of a Seaglider to compensate for drift from currents using depth averaged currents over multiple dives. Those important works focus on adaptive control of vehicles based on current conditions to improve sampling. We instead look at other hydrographic properties in order to optimize sampling of a specific feature.

A number of near real-time feature tracking methods exist for applications such as thermoclines [Cruz and Matos, 2010; Sun et al., 2016; Zhang et al., 2010] and oil spills [Zhang et al., 2011]. These approaches focus on tracking a one-dimensional feature using a single vehicle, while we utilize multiple vehicles to track a two-dimensional feature. Flexas et al. [2018] uses an ocean model and autonomous planning to optimize sampling of submesoscale structures. Our approach focuses on frontal tracking using trailing in-situ vehicle data as apposed to an ocean model.

Other work has investigated two-dimensional feature tracking. Zhang et al. [2013, 2016] utilize the VTHI front detection method on a single vehicle to detect and track an upwelling front on a zig-zag track with a fixed turn angle. Cruz and Matos [2014] tracks any gradient boundary using a single vehicle following a dynamic zig-zag pattern and a lateral gradient detection algorithm to estimate the gradient boundary using an arc whose curvature is defined by the last three front-crossing locations. A similar method can also be applied to tracking the center of a phytoplankton bloom patch [Godin et al., 2011]. Machine learning, in the form of policy learning, has also been applied to the problem of tracking the edge of a harmful algal bloom [Magazzeni et al., 2014]. Other work focuses on tracking algal blooms by flying formations relative to the bloom as tracked by a drifter [Das et al., 2012]. Petillo, Schmidt, and Balasuriya [2012] uses a simulated network of AUVs in order to estimate the boundary of a simulated plume. These all differ from our approach in that we are using multiple vehicles in order to estimate the position and orientation of an ocean front using a method of gridded front detections as well as a linear front model.

## Future Work

### On-Board Planning

On-board planning can eliminate the constraints imposed by off-board planning. The first option is for all vehicle planning to be performed on-board with all information required for coordination relayed through a centralized off-board server. In the case of our planning method, this would involve sending the front detection locations to a centralized server and sending the front location and orientation to each vehicle from the centralized server. This allows for shorter surfacing windows, use of the full dataset, and real time use of the scientific data. However, less processing power is available for the planning and execution software. An updated front detection method could be required depending on the constraints of on-board processing.

The second option removes the use of a centralized shore-based server for vehicle coordination, instead opting for a peer-to-peer based architecture. This requires a method of inter-vehicle communication such as an acoustic modem, limiting the distance vehicles can be from one another. By performing all planning and execution operations on-board the vehicle, surfacing times can be drastically reduced or the vehicles can operate in areas where surfacing is not always possible, such as an ice-covered environment. Real-time planning and coordination is also possible with this method by removing the need for vehicles to surface for communication. The most appropriate paradigm for planning and execution depends the requirements of the planning method itself.

### Front Detection

Throughout this experiment, multiple points of improvement were identified in regards to lateral gradient front detection. Front detection could be improved by gridding data based on distance traveled as opposed to time. This is particularly important for slower moving vehicles such as underwater gliders. The gridding process itself could also be improved by using objective mapping. In this experiment temperature was used, other ocean properties such as, buoyancy could also be used. The lateral gradient front detection method consists of many parameters, a more in-depth analysis of the effects of these parameters would be beneficial. Our front-crossing detection technique could be extended in order to select a crossing based on a set of criteria such as front direction (i.e. cold-to-warm versus warm-to-cold), gradient strength, and front size. By using these different properties a specific front can be targeted.

## Conclusion

This work presents a planning and execution system for a heterogeneous fleet of underwater vehicles and demonstrates it with a method of adaptive control using multiple autonomous underwater vehicles in order to track an ocean front evolving over time. This method utilizes an off-board planner for near real-time front detection, ocean front estimation using a linear model, and vehicle retasking. We build upon the prior efforts of the AOSN deployments and takes a further step towards a fully-autonomous adaptive sampling framework [Thompson et al., 2017].

The experiment was conducted in May and June, 2017 in and around Monterey Bay, California. Three types vehicles were used, two Tethys-Class Long-Range AUVs, two short-range Iver AUVs, and one autonomous underwater glider, a Seaglider. During this experiment we demonstrated the performance of the lateral gradient front detection method on data from all three vehicles and the capability of the autonomous control method for front tracking. We showed that this method is both suitable for a multi-vehicle approach with a dynamic front position and orientation and a single-vehicle approach utilizing a fixed front orientation. The multi-vehicle approach allows for improved synopticity over a zig-zag method when sampling a front. While the use of off-board planning algorithms provides more processing power and allows for flexible implementation for different platforms.

**Acknowledgments** The following work was done under the framework of the Keck Institute for Space Studies (KISS)-funded project "Science-driven Autonomous and Heterogeneous Robotic Networks: A Vision for Future Ocean Observations"[Thompson et al., 2017]. Portions of this work were funded by the Keck Institute and Woods Hole Oceanographic Institution. Portions of this work were performed by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

## References


Bellingham, J. G.; Zhang, Y.; Kerwin, J. E.; Erikson, J.; Hobson, B.; Kieft, B.; Godin, M.; McEwen, R.; Hoover, T.; Paul, J.; et al. 2010. Efficient propulsion for the tethys long-range autonomous underwater vehicle. In *Autonomous Underwater Vehicles (AUV), 2010 IEEE/OES*, 1–7. IEEE.

Branch, A.; Flexas, M. M.; Claus, B.; Clark, E. B.; Thompson, A. F.; Chien, S.; Kinsey, J. C.; Fratantoni, D. M.; Zhang, Y.; Kieft, B.; Hobson, B.; and Chavez, F. P. 2018. Front delineation and tracking with multiple underwater vehicles. *J. Field Robotics* in review.

Crowell, J. 2006. Small auv for hydrographic applications. In *OCEANS 2006*, 1–6.

Cruz, N. A., and Matos, A. C. 2010. Adaptive sampling of thermoclines with autonomous underwater vehicles. In *OCEANS 2010*, 1–6. IEEE.

Cruz, N. A., and Matos, A. C. 2014. Autonomous tracking of a horizontal boundary. In *Oceans-St. John's, 2014*, 1–6. IEEE.

Curtin, T. B., and Bellingham, J. G. 2009. Progress toward autonomous ocean sampling networks. *Deep Sea Research Part II: Topical Studies in Oceanography* 56(3):62 – 67. AOSN II: The Science and Technology of an Autonomous Ocean Sampling Network.

Curtin, T. B.; Bellingham, J. G.; Catipovic, J.; and Webb, D. 1993. Autonomous oceanographic sampling networks. *Oceanography* 6(3):86–94.

Das, J.; Py, F.; Maughan, T.; OReilly, T.; Messié, M.; Ryan, J.; Sukhatme, G. S.; and Rajan, K. 2012. Coordinated sampling of dynamic oceanographic features with underwater vehicles and drifters. *The International Journal of Robotics Research* 31(5):626–646.

Eriksen, C. C.; Osse, T. J.; Light, R. D.; Wen, T.; Lehman, T. W.; Sabin, P. L.; Ballard, J. W.; and Chiodi, A. M. 2001. Seaglider: A long-range autonomous underwater vehicle for oceanographic research. *IEEE J. Oceanic Eng.* 26:424436.

Flexas, M. M.; Troesch, M. I.; Chien, S.; Thompson, A. F.; Chu, S.; Branch, A.; Farrara, J. D.; and Chao, Y. 2018. Autonomous sampling of ocean submesoscale fronts with ocean gliders and numerical model forecasting. *Journal of Atmospheric and Oceanic Technology* 35(3):503–521.

Godin, M. A.; Zhang, Y.; Ryan, J. P.; Hoover, T. T.; and Bellingham, J. G. 2011. Phytoplankton bloom patch center localization by the tethys autonomous underwater vehicle. In *OCEANS'11 MTS/IEEE KONA*, 1–6.

Haley, P.; Lermusiaux, P.; Robinson, A.; Leslie, W.; Logoutov, O.; Cossarini, G.; Liang, X.; Moreno, P.; Ramp, S.; Doyle, J.; Bellingham, J.; Chavez, F.; and Johnston, S. 2009. Forecasting and reanalysis in the monterey bay/california current region for the autonomous ocean sampling network-ii experiment. *Deep Sea Research Part II: Topical Studies in Oceanography* 56(3):127 – 148. AOSN II: The Science and Technology of an Autonomous Ocean Sampling Network.

Hickey, B. M. 1979. The california current system-hypotheses and facts. *Prog. Oceanogr.* 8:191–279.

Hobson, B. W.; Bellingham, J. G.; Kieft, B.; McEwen, R.; Godin, M.; and Zhang, Y. 2012. Tethys-class long range auvs-extending the endurance of propeller-driven cruising auvs from days to weeks. In *Autonomous Underwater Vehicles (AUV), 2012 IEEE/OES*, 1–8. IEEE.

Leonard, N. E.; Paley, D. A.; Lekien, F.; Sepulchre, R.; Fratantoni, D. M.; and Davis, R. E. 2007. Collective motion, sensor networks, and ocean sampling. *Proceedings of the IEEE* 95(1):48–74.

Leonard, N. E.; Paley, D. A.; Davis, R. E.; Fratantoni, D. M.; Lekien, F.; and Zhang, F. 2010. Coordinated control of an underwater glider fleet in an adaptive ocean sampling field experiment in monterey bay. *Journal of Field Robotics* 27(6):718–740.

Lynn, R. J., and Simpson, J. J. 1987. The california current system: The seasonal variability of its physical characteristics. *J. Geophys. Res.* 92:12947–12966.

Magazzeni, D.; Py, F.; Fox, M.; Long, D.; and Rajan, K. 2014. Policy learning for autonomous feature tracking. *Autonomous Robots* 37(1):47–69.

Molemaker, M. J.; McWilliams, J. C.; and Dewar, W. K. 2015. Submesoscale instability and generation of mesoscale anticyclones near a separation of the california undercurrent. *J. Phys. Oc.* 45:613–629.

Monterey Bay Aquarium Research Institute. 2017. Canon spring 2017 expedition.

Petillo, S.; Schmidt, H.; and Balasuriya, A. 2012. Constructing a distributed auv network for underwater plume-tracking operations. *International Journal of Distributed Sensor Networks* 2012:Article ID 191235, 12pp.

Ramp, S.; Davis, R.; Leonard, N.; Shulman, I.; Chao, Y.; Robinson, A.; Marsden, J.; Lermusiaux, P.; Fratantoni, D.; Paduan, J.; Chavez, F.; Bahr, F.; Liang, S.; Leslie, W.; and Li, Z. 2009. Preparing to predict: The second autonomous ocean sampling network (aosn-ii) experiment in the monterey bay. *Deep Sea Research Part II: Topical Studies in Oceanography* 56(3):68 – 86. AOSN II: The Science and Technology of an Autonomous Ocean Sampling Network.

Sun, L.; Li, Y.; Yan, S.; Wang, J.; and Chen, Z. 2016. Thermocline tracking using a portable autonomous underwater vehicle based on adaptive threshold. In *OCEANS 2016-Shanghai*, 1–4. IEEE.

*REFERENCES*

Thompson, A. F.; Chao, Y.; Chien, S.; Kinsey, J.; Flexas, M. M.; Erickson, Z. K.; Farrara, J.; Fratantoni, D.; Branch, A.; Chu, S.; Troesch, M.; Claus, B.; and Kepper, J. 2017. Satellites to seafloor: Toward fully autonomous ocean sampling. *Oceanography* 30(2):160–168.

Troesch, M.; Chien, S. A.; Chao, Y.; and Farrara, J. D. 2016. Planning and control of marine floats in the presence of dynamic, uncertain currents. In *International Conference on Automated Planning and Scheduling*, 431–440.

Zhang, Y.; Bellingham, J. G.; Godin, M.; Ryan, J. P.; McEwen, R. S.; Kieft, B.; Hobson, B.; and Hoover, T. 2010. Thermocline tracking based on peak-gradient detection by an autonomous underwater vehicle. In *OCEANS 2010*, 1–4. IEEE.

Zhang, Y.; McEwen, R. S.; Ryan, J. P.; Bellingham, J. G.; Thomas, H.; Thompson, C. H.; and Rienecker, E. 2011. A peak-capture algorithm used on an autonomous underwater vehicle in the 2010 gulf of mexico oil spill response scientific survey. *Journal of Field Robotics* 28(4):484–496.

Zhang, Y.; Bellingham, J. G.; Ryan, J. P.; Kieft, B.; and Stanway, M. J. 2013. Two-dimensional mapping and tracking of a coastal upwelling front by an autonomous underwater vehicle. *Proc. MTS/IEEE Oceans'13* 1–4.

Zhang, Y.; Bellingham, J. G.; Ryan, J. P.; Kieft, B.; and Stanway, M. J. 2016. Autonomous four-dimensional mapping and tracking of a coastal upwelling front by an autonomous underwater vehicle. *Journal of Field Robotics* 33(1):67–81.

# Knowledge Representation and Interactive Learning of Domain Knowledge for Human-Robot Interaction

**Mohan Sridharan**
School of Computer Science
University of Birmingham, UK
m.sridharan@bham.ac.uk

**Benjamin Meadows**
Electrical and Computer Engineering
The University of Auckland, NZ
bmea011@aucklanduni.ac.nz

**Abstract**

This paper describes an integrated architecture for representing, reasoning with, and interactively learning domain knowledge in the context of human-robot collaboration. Specifically, Answer Set Prolog, a declarative language, is used to represent and reason with incomplete commonsense knowledge about the domain. Non-monotonic logical reasoning identifies knowledge gaps and guides the interactive learning of relations that represent actions, and of axioms that encode affordances and action preconditions and effects. Learning uses probabilistic models of uncertainty, and observations from active exploration, reactive action execution, and human (verbal) descriptions. The learned actions and axioms are used for subsequent reasoning. The architecture is evaluated on a simulated robot assisting humans in an indoor domain.

## 1 Introduction

Consider one or more robots[1] assisting humans in an office or a home, e.g., delivering desired objects or guiding people to particular locations. Information about such domains often includes commonsense knowledge, especially default knowledge that holds in all but a few exceptional circumstances, e.g., "books are usually in the library, but cookbooks may be in the kitchen". Domain knowledge may also include some understanding of action preconditions and effects, and action capabilities, i.e., affordances. Human participants will, however, lack the time and expertise to provide comprehensive domain information or elaborate feedback. Robots will thus need to reason with incomplete domain knowledge and revise this knowledge over time. The architecture described in this paper is a step towards addressing these open problems; it is based on the following tenets:

- Knowledge elements encode symbolical content about object constants, relations representing domain attributes and actions at different levels of abstraction, and axioms composed of these relations.
- Knowledge elements are revised non-monotonically by reasoning with knowledge and observed outcomes of actions that may be immediate or delayed.
- Affordances are defined jointly over the attributes of agents and objects in the context of particular actions.
- Reasoning, learning and interaction are coupled; values of state-action pairs are revised using observations obtained from active exploration and reactive action execution.

The combination of these tenets is novel, and we implement them using the complementary strengths of declarative programming, probabilistic reasoning, and relational learning through induction and reinforcement. In this paper, we focus on the interplay between reasoning and learning, and abstract away some aspects of our overall architecture, e.g., we flatten some levels of the representation and do not describe probabilistic modeling of perceptual uncertainty. Instead, we describe the following key capabilities:

- Incomplete domain knowledge described in an action language is translated into a relational representation in Answer Set Prolog (ASP) for inference, planning and diagnostics. ASP-based reasoning also automatically limits interactive learning to the relevant part of the domain.
- Previously unknown actions' names, preconditions, effects, and objects over which they operate, along with associated affordances, are learned using decision-tree induction and relational reinforcement learning based on observations of active exploration, reactive action execution, and verbal cues from humans.

We evaluate these capabilities in the context of a simulated robot delivering objects to particular people or locations in an indoor domain. We first describe the proposed architecture and algorithm (Section 2), followed by some results of experimental evaluation (Section 3). Then, Section 4 reviews related work, followed by a description of conclusions and future work in Section 5.

## 2 Proposed Architecture

Figure 1 depicts key components of the overall architecture. Incomplete domain knowledge is encoded in an action language to construct tightly-coupled relational representations at two resolutions. For any given goal, reasoning with commonsense knowledge at the coarse resolution provides a sequence of abstract actions. Each abstract action is implemented as a sequence of concrete actions by a partially observable Markov decision process (POMDP) that reasons probabilistically over the relevant part of the fine-resolution representation, with action outcomes and observations updating the coarse-resolution history. As stated

[1] We use "robot", "agent", and "learner" interchangeably.

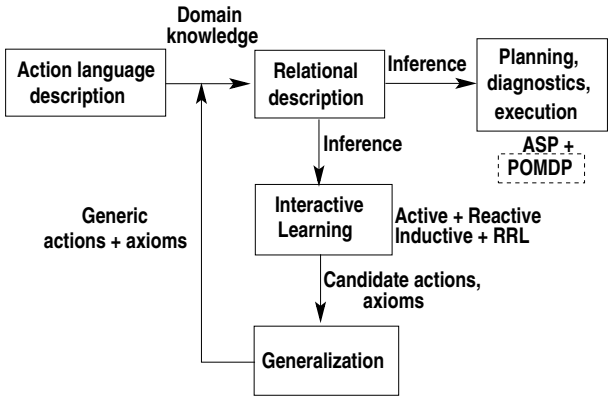



Figure 1: Architecture combines complementary strengths of declarative programming, probabilistic reasoning, and interactive learning for reasoning with and learning domain knowledge.

earlier, we abstract away the reasoning at different resolutions and the probabilistic modeling of perceptual uncertainty, and focus on the interplay between representation, reasoning, and learning. The relational representation is thus translated into an ASP program for planning and diagnostics. ASP-based reasoning also guides the interactive learning of actions, affordances, and the preconditions and effects of actions. This learning uses observations obtained through active exploration, reactive execution, and human (verbal) descriptions—the learned knowledge is used for subsequent reasoning. We use the following domain to illustrate our architecture's capabilities.

**Example 1.** *[Robot Assistant (RA) Domain]* A simulated robot/learner finds, labels, and delivers objects to people or places (office, kitchen, library, workshop) in a building. Each place may have one or more instances of objects such as desk, book, cup and computer. Each human has a particular role (e.g., engineer, manager, salesperson). Objects are characterized by weight (heavy, light), surface (brittle, hard), status (intact, damaged), and labeled (true, false). The robot's arm has a type (electromagnetic, pneumatic). The actions available to the robot include pickup, putdown, move, label, and serve, but it may not know about some actions or axioms (i.e., rules) governing domain dynamics such as:

- A pneumatic arm cannot be used to serve a brittle object.
- Serving an object to a salesperson causes it to be labeled.
- An object with a brittle surface cannot be labeled unless the robot has an electromagnetic arm.

There may be other robots that (are assumed, for simplicity, to) have identical capabilities and cannot communicate with the learner. Humans and the learner can observe these robots. Humans can verbally describe other robots' activities, e.g., "Robot labeled the hard, hefty item" to help the learner acquire knowledge of previously unknown actions and axioms. Although this domain may appear simplistic, it becomes complex as the number of ground instances of objects and their attributes increases, e.g., there were $\approx 18,000$ combinations of ground static attributes and $\approx 11$ million combinations of ground fluent terms in an instantiation.

## 2.1 Knowledge Representation and Reasoning

We first describe the action language encoding of domain dynamics, and its translation to CR-Prolog programs for knowledge representation and reasoning.

**Action Language** Action languages are formal models of parts of natural language used for describing transition diagrams of dynamic systems. We use action language $\mathcal{AL}_d$ (Gelfond and Inclezan 2013) to describe the transition diagrams at different resolutions. $\mathcal{AL}_d$ has a sorted signature with *statics*, *fluents* and *actions*. Statics are domain attributes whose truth values cannot be changed by actions, whereas fluents are domain attributes whose truth values can be changed by actions. Fluents can be *basic* or *defined*. Basic fluents obey the laws of inertia and can be changed by actions. Defined fluents do not obey the laws of inertia and are not changed directly by actions—their values depend on other fluents. Actions are defined as a set of elementary operations. A domain attribute *p* or its negation $\neg p$ is a *literal*. $\mathcal{AL}_d$ allows three types of statements:

$a$ **causes** $l_b$ **if** $p_0, \ldots, p_m$ (Causal law)
$l$ **if** $p_0, \ldots, p_m$ (State constraint)
**impossible** $a_0, \ldots, a_k$ **if** $p_0, \ldots, p_m$ (Executability condition)

where $a$ is an action, $l$ is a literal, $l_b$ is a basic literal, and $p_0, \ldots, p_m$ are domain literals.

**Domain Representation: Signature and Axioms** The domain representation consists of system description $\mathcal{D}$, which is a collection of statements of $\mathrm{AL}_d$, and history $\mathcal{H}$. $\mathcal{D}$ has a sorted signature $\Sigma$ and axioms that describe the transition diagram $\tau$. $\Sigma$ defines the basic sorts, and domain attributes and actions. Basic sorts of the RA domain include place, robot, role, book, weight, status etc, which are arranged hierarchically, and sort step for temporal reasoning. $\Sigma$ includes ground instances of sorts, e.g., {office, workshop, kitchen, library} of sort place. Domain attributes and actions are described in terms of the sorts of their arguments. The RA domain has fluents such as loc(entity, place), the location of the robot and objects, with the locations of humans and other robots (if any) modeled as defined fluents whose values are obtained from external sensors; and in_hand(robot, object), which denotes whether a particular object is in the robot's hand. Static attributes include arm_type(robot, type), obj_status(object, status) etc, and actions include move(robot, place), pickup(robot, object), and serve(robot, object, person). The representation also includes a relation holds(fluent, step) that implies a particular fluent is true at a particular timestep.

Axioms of the RA domain include causal laws, state constraints and executability conditions such as:

$\mathrm{move}(\mathrm{rob}_1, \mathrm{L})$ **causes** $\mathrm{loc}(\mathrm{rob}_1, \mathrm{L})$
$\mathrm{serve}(\mathrm{rob}_1, \mathrm{O}, \mathrm{P})$ **causes** $\mathrm{in\_hand}(\mathrm{P}, \mathrm{O})$
$\mathrm{loc}(\mathrm{O}, \mathrm{L})$ **if** $\mathrm{loc}(\mathrm{rob}_1, \mathrm{L}), \mathrm{in\_hand}(\mathrm{rob}_1, \mathrm{O})$
**impossible** $\mathrm{pickup}(\mathrm{rob}_1, \mathrm{O})$ **if** $\mathrm{loc}(\mathrm{rob}_1, \mathrm{L}_1), \mathrm{loc}(\mathrm{O}, \mathrm{L}_2), \mathrm{L1} \neq \mathrm{L2}$

The history $\mathcal{H}$ of a dynamic domain is usually a record of fluents observed to be true or false at a particular time step, i.e., $\mathtt{obs}(\mathtt{fluent}, \mathtt{boolean}, \mathtt{step})$, and the occurrence of an action at a particular time step, i.e., $\mathtt{occurs}(\mathtt{action}, \mathtt{step})$. This notion was expanded to represent defaults describing the values of fluents in the initial state (Sridharan et al. 2017), e.g., "books are usually in the library and if not there, they are normally in the office" is encoded as:

$$\textbf{initial default } \mathtt{loc}(X, \mathtt{library}) \textbf{ if } \mathtt{book}(X)$$
$$\textbf{initial default } \mathtt{loc}(X, \mathtt{office}) \textbf{ if } \mathtt{book}(X), \neg \mathtt{loc}(X, \mathtt{library})$$

We can also encode exceptions to these defaults, e.g., "cookbooks are in the kitchen".

**Domain Representation: Affordances** We define affordances, i.e., action capabilities, as relations between attributes of robot(s) and object(s) in the context of particular actions. Negative (i.e., forbidding or dis-) affordances describe unsuitable combinations of objects, robots, and actions. Positive affordances describe permissible uses of objects in actions by agents, including exceptions to executability conditions that prevent the use of the corresponding action during planning. In $\mathcal{AL}_d$, we represent affordances in a distributed manner, as follows:

$$\textbf{impossible } A \textbf{ if } \mathtt{aff\_forbids}(\mathrm{ID}, A)$$
$$\mathtt{aff\_forbids}(\mathtt{id}_i, A) \textbf{ if } \ldots$$
$$\textbf{impossible } A \textbf{ if } \ldots, \mathtt{not}\ \mathtt{aff\_permits}(\mathrm{ID}, A)$$
$$\mathtt{aff\_permits}(\mathtt{id}_j, A) \textbf{ if } \ldots$$

The first two statements say that action $A$ cannot occur if it is not afforded, and specify the conditions (i.e., attributes of robot and object) under which the action is not afforded. The last two statements say that an action $A$ that is not considered during planning due to an executability condition may have a positive affordance as an exception, and define the positive affordance. Each action can have multiple affordances indexed by the *id*s. This representation of knowledge improves generalization, and can simplify inference.

**ASP-based inference** The domain representation is translated into a program $\Pi(\mathcal{D}, \mathcal{H})$ in CR-Prolog[2], a variant of ASP that incorporates consistency restoring (CR) rules (Balduccini and Gelfond 2003). ASP is based on stable model semantics, and supports *default negation* and *epistemic disjunction*, e.g., unlike "$\neg a$" that states *a is believed to be false*, "$\mathtt{not}\ a$" only implies *a is not believed to be true*, and unlike "$p \vee \neg p$" in propositional logic, "$p$ or $\neg p$" is not tautologous. ASP can represent recursive definitions and constructs that are difficult to express in classical logic formalisms, and it supports non-monotonic logical reasoning, i.e., the ability to revise previously held conclusions based on new evidence. The program $\Pi$ includes the signature and axioms of $\mathcal{D}$, inertia axioms, reality checks, and observations, actions, and defaults from $\mathcal{H}$. Every default also has a CR rule that allows the robot to assume the default's conclusion is false under exceptional circumstances, to restore consistency. Each *answer set* of an ASP program represents the set of beliefs of an agent associated with the program. Algorithms for computing the entailment, and for planning and diagnostics, reduce these tasks to computing answer sets of CR-Prolog programs.

Reasoning with incomplete or incorrect knowledge may overlook valid plans, find suboptimal plans, or provide plans whose execution has unintended outcomes. For instance, the robot in the RA domain is asked to deliver textbook $\mathtt{book}_1$ to the $\mathtt{office}$. It uses default knowledge to compute the plan $\mathtt{move}(\mathtt{rob}_1, \mathtt{library})$, $\mathtt{pickup}(\mathtt{rob}_1, \mathtt{book}_1)$, $\mathtt{move}(\mathtt{rob}_1, \mathtt{office})$, $\mathtt{putdown}(\mathtt{rob}_1, \mathtt{book}_1)$. This does not succeed because (unknown to the robot) its electromagnetic arm cannot pick up the heavy book. We next describe the interactive learning of such unknown knowledge.

## 2.2 Interactive Learning

Obtaining labeled samples to learn previously unknown actions, and axioms is difficult in complex domains, and humans may have limited time and expertise. Also, the effects of actions may be observed immediately or after a delay. We thus enable the robot to interactively acquire labeled examples. To speed up learning and to simulate learning without running many trials on a robot, we introduce two schemes: (i) active learning from verbal cues provided by humans; (ii) relational reinforcement learning based on observations from active exploration or reactive action execution. We describe these schemes below.

**Learning from Human Interaction** To acquire domain knowledge from the verbal cues provided by humans describing the observed behaviors of other robots, the learner makes the following assumptions:

- Other robots have the same capabilities as the learner;
- Learner can generate logic statements corresponding to attributes of robot(s) or object(s) in the observed action;
- Humans correctly describe one activity at a time.

These assumptions are reasonable for many robotics domains, and simplify interaction with humans.

The learner solicits human input when available and receives a transcribed verbal description of an action and observations of the action's consequences, e.g., the learner may receive "The robot is labeling the fairly big textbook." and $\mathtt{labeled}(\mathtt{book}_1)$. We use the Stanford log-linear part-of-speech (POS) tagger (Toutanova et al. 2003). We employ a left, second-order sequence information model to determine each word's POS tag and append it to the word. In our example, the output is a string such as "The_DT robot_NN is_VBZ labeling_VBG the_DT fairly_RB big_JJ textbook_NN", where "VB" represents a verb, "NN" is a noun etc. The learner transforms this string to <word, POS> pairs, and transforms the sentence's verb into first-person present-tense using rules from a lemma list (Someya 1998) and WordNet (Miller 1995), e.g., $< \mathtt{is}, \mathtt{VBZ} > < \mathtt{labeling}, \mathtt{VBG} >$ becomes the verb "label". The learner also marks each noun phrase as a sequence of zero or more adjectival terms followed by a noun, discarding other interleaved words. Our example sentence's noun phrases are

[2]We use the terms "ASP" and "CR-Prolog" interchangeably.

*robot* and *big textbook*. Nouns signify object sorts and adjectival terms signify values of static attributes. To determine terms' referents, WordNet relations such as *linked synsets* are used to find a synonym that is also a domain symbol, e.g., "big" and "heavy" share a WordNet synset, *heavy* is an attribute value, and $\mathtt{book}(\mathtt{book}_1)$ and $\mathtt{obj\_weight}(\mathtt{book}_1, \mathtt{heavy})$ are domain attributes. The matched domain symbols combine to refer to particular objects. We require static attributes' values to be disjoint sets, and each noun phrase to signify an existing object—these are true by design in our domain.

The robot constructs a literal for the action from the verb and the object referents, e.g., $\mathtt{label}(\mathtt{rob}_1, \mathtt{book}_1)$. The arguments' lowest-level sorts are assumed to be the valid arguments, e.g., $\mathtt{label}(\#\mathtt{robot}, \#\mathtt{book})$. If this candidate action does not match any known action literal, the robot lifts the literal, its arguments and the observed action consequences. This forms the basis for constructing candidate causal laws and generalizing over time. For instance:

$$\mathtt{label}(\mathtt{rob}_1, \mathtt{book}_1) \ \textbf{causes} \ \mathtt{labeled}(\mathtt{book}_1)$$

is lifted to:

$$\mathtt{label}(R, B) \ \textbf{causes} \ \mathtt{labeled}(B)$$

If, on the other hand, the new literal matches an existing one, the first common ancestor of each argument's sort is found. For instance, if the learner knows $\mathtt{label}(\#\mathtt{robot}, \#\mathtt{cup})$ and finds that $\mathtt{label}(\#\mathtt{robot}, \#\mathtt{book})$ has matching consequences, it will generalize to $\mathtt{label}(\#\mathtt{robot}, \#\mathtt{object})$. This method for learning from interaction with humans adapts existing natural language processing methods to work with our representation. It helps the learner acquire a previously unknown action's name, and the sorts of objects the action operates on. However, this knowledge is not sufficient because the learner may still not know axioms that govern the domain dynamics related to this action. This missing knowledge is acquired using the second learning scheme below.

**Relational Reinforcement Learning** The second learning scheme enables axiom discovery by active exploration of the transition corresponding to a particular action, or by exploration in response to unexpected and unexplained transitions. To explore a particular transition, the resultant state is set as the goal of a reinforcement learning (RL) problem, i.e., the objective is to find state-action pairs most likely to lead to this (and other similar) states. The underlying MDP is defined by a set of states ($S$), set of actions ($A$), state transition function $T_f : S \times A \times S' \rightarrow [0, 1]$, and reward function $R_f : S \times A \times S' \rightarrow \mathfrak{R}$. Similar to classical RL formulations, $T_f$ and $R_f$ are unknown to the agent. Each state has ground atoms formed of the domain attributes (i.e., fluent terms and statics), and a boolean literal describing whether the most recent action had the expected outcome. Each action is a ground action of the system description. $T_f$ and $R_f$ are constructed from statistics collected in an initial training phase; $T_f$ is a probabilistic model of the uncertainty in state transitions, while $R_f$ provides instantaneous rewards for executing particular actions in particular states. The RL formulation is constructed automatically from the system description—(Sridharan et al. 2017) describes a method for translating an ASP-based system description to a representation for probabilistic sequential decision making.

The values of state-action pairs are estimated in a series of episodes, until convergence, using the Q-learning algorithm (Sutton and Barto 1998). In each episode, the agent executes a sequence of actions chosen using an $\epsilon$-greedy algorithm and eligibility traces. The combinations of states and actions invalidated by existing axioms are not explored. Each episode terminates when a time limit is exceeded or the target action succeeds. The physical configuration of objects is then reset to its state from the beginning of the episode, and a new episode begins. Such a formulation can become computationally intractable for complex domains. A key advantage of our architecture is that ASP-based reasoning can be used to automatically restrict the object constants, domain attributes and axioms *relevant* to the desired transition, i.e., to those that influence or are influenced by the transition, significantly reducing the search space. This notion of relevance is based on the following desiderata regarding the relations that may appear in a discovered axiom:

- For any static attribute that may exist in the body of the discovered axiom, we wish to explore all possible elements in the range of the attribute, e.g., for action $\mathtt{serve}(\mathtt{rob}_1, \mathtt{cup}_1, \mathtt{person}_1)$, all possible weights of $\mathtt{cup}_1$ and roles of $\mathtt{person}_1$ are explored.
- For any fluent that may appear in the body of the axiom, we wish to explore only those elements in the range of the fluent that occur in the state before or after the state transition. Any other element cannot, by design, be influenced by this transition anyway.

ASP-based reasoning is used to encode these requirements and automatically construct the system description $\mathcal{D}(T)$, the part of $\mathcal{D}$ relevant to the transition $T$. To do so, we first define the object constants relevant to the transition of interest. These definitions are adapted from the definitions introduced in (Sridharan et al. 2017).

**Definition 1.** *[Relevant object constants]*
Let $a_{tg}$ be the *target action* that when executed in state $\sigma_1$ resulted in the unexpected transition $T = \langle \sigma_1, a_{tg}, \sigma_2 \rangle$. Let $\mathtt{relCon}(T)$ be the set of object constants of $\Sigma$ of $\mathcal{D}$ identified using the following rules:

1. Object constants from $a_{tg}$ are in $\mathtt{relCon}(T)$;
2. If $f(x_1, \ldots, x_n, y)$ is a literal formed of a domain attribute, and the literal belongs to $\sigma_1$ or $\sigma_2$, but not both, then $x_1, \ldots, x_n, y$ are in $\mathtt{relCon}(T)$;
3. If the body $B$ of an axiom of $a_{tg}$ contains an occurrence of $f(x_1, \ldots, x_n, Y)$, a term whose domain is ground, and $f(x_1, \ldots, x_n, y) \in \sigma_1$, then $x_1, \ldots, x_n, y$ are in $\mathtt{relCon}(T)$.

Constants from $\mathtt{relCon}(T)$ are said to be *relevant* to $T$, e.g., for action $a_{tg} = \mathtt{serve}(\mathtt{rob}_1, \mathtt{cup}_1, \mathtt{person}_1)$ in the RA domain, with $\mathtt{loc}(\mathtt{rob}_1, \mathtt{office})$, $\mathtt{loc}(\mathtt{cup}_1, \mathtt{office})$, and $\mathtt{loc}(\mathtt{person}_1, \mathtt{office})$ in $\sigma_1$, the relevant object constants include $\mathtt{rob}_1$, $\mathtt{cup}_1$, $\mathtt{person}_1$, and $\mathtt{office}$.

**Definition 2.** *[Relevant system description]*
The system description relevant to the desired transition $T =$

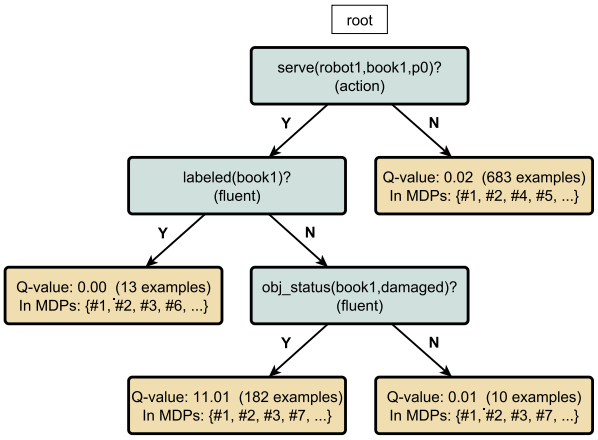



Figure 2: Illustrative example of a Binary Decision Tree (BDT) with nodes representing tests of domain literals. The BDT is constructed incrementally over time.

$\langle \sigma_1, a_{tg}, \sigma_2 \rangle$, i.e., $\mathcal{D}(\mathsf{T})$, is defined by signature $\Sigma(\mathsf{T})$ and axioms. $\Sigma(\mathsf{T})$ is constructed to comprise the following:

1. Basic sorts of $\Sigma$ that produce a non-empty intersection with $\mathtt{relCon}(\mathsf{T})$.
2. All object constants of basic sorts of $\Sigma(\mathsf{T})$ that form the range of a static attribute.
3. The object constants of basic sorts of $\Sigma(\mathsf{T})$ that form the range of a fluent, or the domain of a fluent or a static, and are in $\mathtt{relCon}(\mathsf{T})$.
4. Domain attributes restricted to basic sorts of $\Sigma(\mathsf{T})$.

Axioms of $\mathcal{D}(\mathsf{T})$ are those of $\mathcal{D}$ restricted to $\Sigma(\mathsf{T})$. For $a_{tg} = serve(\mathtt{rob}_1, \mathtt{cup}_1, \mathtt{person}_1)$ in our current example, $\mathcal{D}(\mathsf{T})$ does not include other robots, cups or people in the domain. It can be shown that for each transition in the transition diagram of the system description $\mathcal{D}$, there is a transition in the transition diagram of $\mathcal{D}(\mathsf{T})$. States of $\mathcal{D}(\mathsf{T})$, i.e., literals formed of fluents and statics in the answer sets of the ASP program, are states in the RL formulation, and actions are ground actions of $\mathcal{D}(\mathsf{T})$. Furthermore, it is possible to pre-compute or reuse some of the information used to construct $\mathcal{D}(\mathsf{T})$ for any given $\mathsf{T}$.

Once the $\mathcal{D}(\mathsf{T})$ relevant to the target transition has been identified, the RL formulation is constructed as before to compute the values of state-action combinations. The extent to which computing $\mathcal{D}(\mathsf{T})$ reduces the search space depends on the relationships between the domain attributes and axioms. For instance, although there are several thousand static attribute combinations and more than a million object configurations in our instantiation of the RA domain, computing $\mathcal{D}(\mathsf{T})$ often reduces the space of attribute combinations to as few as 12 for the $serve$ action. However, in other domains with complex relationships between objects, exploration may need to be further limited to a fraction of this restricted state space. Furthermore, Q-learning does not generalize to relationally equivalent states.

Inspired by the RRL-TG algorithm (Driessens and Ramon 2003), we facilitate generalization to relationally equivalent states by constructing a binary decision tree (BDT) whose nodes represent tests of domain literals—Figure 2 shows an example BDT. Unlike the destructive branching of RRL-TG, we model the partial description of a state-action pair as a path to a leaf where we store the remaining state information. When Q-value variance is reduced by adding a test at a leaf, the BDT is expanded and used to compute policies in subsequent RL episodes. To learn generic versions of axioms, the robot explores different values of static attributes and fluent literals. ASP-based reasoning automatically selects relevant combinations to make exploration tractable, and uses sampling if the search space is too large. Unlike traditional RRL methods, the learned Q-values now represent values across different MDPs.

After learned values converge, axioms are constructed from the BDT. A partial description (path to leaf) is selected if it is associated with the high accrued value, and all subsets of its literals become candidate axioms. Since each candidate axiom could correspond to different branches of the BDT, the learner randomly draws a number of samples without replacement, considers additional literals stored at the leaves, and alters candidates that match the sample. Candidates with sufficient support are *validated*, i.e., tested under conditions that are simulated to match the transition that triggered learning. Candidates that do not pass these tests are removed from further consideration. For instance, if a learned executability condition is correct, executing the action when literals in the body are true should not provide the expected outcome. Note that these tests are guaranteed to not eliminate any valid axioms although they may not remove all false positive candidates. The final candidates are lifted by replacing ground terms with variables, and added to the ASP program as axioms for subsequent reasoning. We refer to this RRL approach as "Q-RRL".

**Control Loop** Algorithm 1 describes the overall control loop for reasoning and learning in our architecture. The baseline behavior (lines 5-17) is to plan and execute actions to achieve the given goal as long as a consistent model of history is can be computed (lines 7-9). If such a model cannot be constructed, it is attributed to an unexplained, unexpected transition, and the robot triggers Q-RRL (lines 9-12) to discover the corresponding unknown axioms (lines 20-21). If there is no active goal to be achieved, the robot triggers active learning (lines 13-16) using Q-RRL (lines 25-27) or verbal descriptions obtained from a human participant (lines 23-25) to learn previously unknown actions or axioms. When in the learning mode, the robot can be interrupted if needed (lines 18-19), e.g., to pursue a new goal.

## 3 Experimental Setup and Results

In this section, we describe the results of experimentally evaluating the following hypotheses:

- **H1**: Active learning of actions from verbal descriptions provides a foundation for further learning;
- **H2**: Q-RRL provides a mechanism for discovering axioms related to an action;
- **H3**: Learned knowledge improves plan quality.

**Algorithm 1:** Overall control loop.

**Input:** $\Pi(\mathcal{D}, \mathcal{H})$; goal description; initial state $\sigma_1$.
**Output:** Control signals for robot to execute.

```
   /* Start with planning */
1  planMode = true, learnType = 0
2  while true do
3      Add observations to history.
4      ComputeAnswerSets(Π(D, H))
5      if planMode then
6          if existsGoal then
               /* Goal exists, consistent
                  model, execute plan */
7              if explainedObs then
8                  ExecutePlanStep()
9              else
                   /* Q-RRL */
10                 planMode = false
11                 learnType = 1
12             end
13         else
               /* Active learning */
14             planMode = false
15             learnType = 2
16         end
17     else
           /* Interrupt learning if needed
              */
18         if interrupt then
19             planMode = true
           /* Continue learning */
20         else if learnType == 1 then
21             ContinueRRL()
22         else if learnType == 2 then
23             if verbalCue then
24                 ContinueActiveLearn()
25             else
26                 ContinueActiveRRL()
27             end
28     end
29 end
```

These hypotheses were evaluated in the RA domain (Example 1) in the context of two actions (serve and label). We considered the following target axioms to be discovered by the robot, for the action serve:

(1) Serving an object to a salesperson causes it to be labelled (*causal law*);

(2) A damaged object cannot be served to a person who is not an engineer (*executability condition*);

(3) A robot with a pneumatic arm cannot serve a brittle object (*negative affordance*); and

(4) A damaged object cannot be served to a person who is not an engineer, unless it is labeled (*positive affordance*).

and for the action label:

(5) An object with a brittle surface cannot be labeled by a robot (*executability condition*);

(6) A damaged object cannot be labeled by a robot with a pneumatic arm (*negative affordance*);

(7) Labelling a light object with a pneumatic arm causes it to be damaged (*causal law*); and

(8) An object with a brittle surface cannot be labelled by a robot, unless the object is heavy and the robot has an electromagnetic arm (*positive affordance*).

We provide execution traces in support of hypothesis H1; hypotheses H2 and H3 are evaluated quantitatively.

### 3.1 Experimental Setup

The initial setup included experimentally setting the values of some parameters in Q-RRL by trading off accurate estimation of policies against processing time, e.g., learning rate and exploration preference were fixed at 0.1. Candidate axioms were constrained to have no more than two positive literals and two negative literals formed of domain attributes—this limit can be increased as needed in other domains at the expense of a corresponding increase in computational complexity. Up to 10 validation tests were conducted to evaluate candidate axioms.

In the experimental trials reported below, the robot learned the representation for each action and associated causal law from verbal descriptions. The robot then used Q-RRL to learn one causal law, one executability condition, one positive affordance and one negative affordance for each of the two actions (serve, label). Axioms for each action can be discovered concurrently. Unless stated otherwise, each value of the performance measures reported below was averaged over 1000 repetitions (e.g., for each axiom). We used *precision* and *recall* as the performance measures. Axioms were required to exactly match the ground truth to be counted as true positives; under-specifications (e.g., some missing literals) and most over-specifications (e.g., unnecessary literals) were considered false positives. Plan quality was measured as the ability to compute a minimal plan to achieve the desired goal.

### 3.2 Execution Trace

The following execution trace supports H1 by illustrating learning of actions and the objects those actions operate on, using verbal cues from human participants.

**Execution Example 1.** *[Learning from human input]*
Suppose the robot in the RA domain (Example 1) does not know that it can label and serve objects, and does not know the related axioms. For each of the actions, we gave the agent five descriptive examples of the action being applied, with descriptions that were grammatically-correct English statements that upheld our assumptions, but otherwise varied arbitrarily. First consider the label action:

- The learner receives "A robot is labeling the lightweight cup", with the observation $\mathtt{labeled}(\mathtt{cup}_1)$. It parses the statement, matches it to the domain, lifts it to store $\mathtt{label}(\#\mathtt{robot}, \#\mathtt{cup})$, and infers:

$$\mathtt{label}(\mathtt{R}, \mathtt{B}) \textbf{ causes } \mathtt{labeled}(\mathtt{B})$$

- Next, the learner receives "Robot labeled one computer", and $\mathtt{labeled}(\mathtt{comp}_1)$. It learns the signature $\mathtt{label}(\#\mathtt{robot}, \#\mathtt{computer})$ and generalizes over the learned signatures to obtain $\mathtt{label}(\#\mathtt{robot}, \#\mathtt{object})$.
- Further input descriptions are automatically reconciled either when specific sorts are subsumed by more general ones, e.g., when it learns from "The pneumatic robot labels the light breakable cup", or the parse results in an exact match for the action description, as in "Next the robot labeled the hard, hefty item".

Next, in the context of learning the *serve* action:

- The learner receives "A robot serves a manual to the manager" and the observation $\mathtt{in\_hand}(\mathtt{p}_1, \mathtt{book}_1)$. It produces the action description $\mathit{serve}(\#\mathtt{robot}, \#\mathtt{book}, \#\mathtt{person})$ and extracts the causal law:

$$\mathit{serve}(\mathsf{R}, \mathsf{O}, \mathsf{P}) \ \textbf{causes} \ \mathtt{in\_hand}(\mathsf{P}, \mathsf{O})$$

- Next, the learner is given "The pneumatic robot is serving the breakable cup to the clerical person over there" and $\mathtt{in\_hand}(\mathtt{p}_0, \mathtt{cup}_1)$. Generalizing over the two examples results in $\mathit{serve}(\#\mathtt{robot}, \#\mathtt{object}, \#\mathtt{person})$. The remaining sentences, "Robot serves ledger to clerical person" and "A robot served a lightweight cup to an expert", fit the inferred structures and do not change them.

For both actions, two examples were sufficient to reach the required level of generality to model the action and an initial causal law. A key advantage of learning from verbal cues is that only a small number of examples are needed to learn the actions and the objects that they operate on. This is especially useful when actions have irreversible effects. The disadvantage is that humans are expected to provide correct descriptions of the behaviors they observe, although the robot can identify and revise any incorrect information learned and included in the ASP program.

It is important to appreciate the benefits of the distributed representation used in the architecture. First, this representation simplifies inference and information reuse. For instance, if a cup has a graspable handle, this relation also holds true for other objects with handles. If an affordance prevents the robot from picking up a heavy object, this information may be used to infer that it cannot open a large window. This relates to research in psychology which indicates that humans can judge action capabilities of others without actually observing them perform the target actions (Ramenzoni et al. 2010). Second, it becomes possible to respond efficiently to queries that require consolidation of knowledge across different attributes of objects or robots, and to develop composite affordance relations, e.g., a hammer may afford an "affix objects" action in the context of a specific agent because the handle affords a pickup action and the hammer affords a swing action, for the agent. Finally, learning from verbal descriptions can be used to provide more meaningful explanations of decisions.

### 3.3 Quantitative Evaluation

We experimentally evaluated hypotheses H2 and H3.

| Action | Recall | Precision | Precision (validated) |
|---|---|---|---|
| label | 0.92 | 0.82 | 0.96 |
| serve | 0.88 | 0.70 | 0.95 |

Table 1: Accuracy when Q-RRL was used to discover multiple axioms corresponding to two specific actions. High recall and precision are attained, especially after candidate axioms are validated.

**H2: Q-RRL enables reliable discovery of axioms** We explored whether Q-RRL can learn new axioms related to a known (or newly learned) action. Results averaged over the four axioms for each action are summarized in Table 1. We observe that Q-RRL attains high recall and precision, especially after the candidate axioms are validated. The accuracy of discovering the axioms corresponding to the *serve* action is a little lower than that for the label action, as it is more complex, i.e., it has more arguments. There were very few differences in the values of performance measures for causal laws, executability conditions and negative affordances. The recall and precision measures were a little lower for positive affordances since axioms corresponding to positive affordances are more complex—they add context to an executability condition to make the corresponding action applicable. Note that human input is not essential for this learning—a robot could learn from experiences accumulated over time.

**H3: Learning improves plan quality** To evaluate hypothesis H3, we explored the effects of the discovered axioms on the system's ability to generate plans that provide the desired outcome. For each axiom of each target action, we conducted 1000 paired ASP-based planning trials with and without the corresponding target axiom in the system description. The trials used randomized scenarios in which the target action was required to achieve the goal.

We found that adding the learned executability conditions or negative affordances resulted in 13% (*serve*) or 23% (label) fewer plans found. Adding the positive affordances resulted in 17% (*serve*) or 23% (label) more plans. These results are expected, as executability conditions and negative affordances preclude actions in some contexts, and knowledge of positive affordances serves to enable particular transitions. We performed additional trials which added or removed all the learnable axioms collectively, resulting in a difference of 19% (*serve*) or 58% (label) in the plans found. Furthermore, we verified that all the plans that were computed after including the target axioms were correct.

In the paired trials that included or excluded the causal laws extracted from the verbal cues, there was no measurable difference in the number of plans found. This is expected; a causal law for *serve* produces outcomes which impact the applicability of other actions, and similarly for label. This will be the case for any scenario in which the plan produced does not repeat the action influenced by the causal law. Given alternative runs that involve planning for a random goal, we observed that the presence or absence of causal laws had an impact on the number of plans found.

Our evaluation also included other findings. For instance, in our experiments, we found that using the ASP-based inference to guide learning makes the learning significantly

more efficient. We also observed that RL with the relational representation significantly speeds up the learning in comparison with not using the relational representation. Finally, we introduced a percentage chance per action to encounter actuator noise during learning to test the system's robustness, and found a steady decline in accuracy, e.g., 0.89 recall and 0.95 validated precision without noise, 0.69 recall and 0.53 validated precision at 10% noise, or 0.56 recall and 0.34 validated precision at 20% noise. Most false positives were merely overly-specific variations of correct axioms.

## 4 Related Work

Agents often have to represent and reason with incomplete domain knowledge, and learn from observations. Early work used a first-order logic representation and incrementally refined the action operators but did not allow for different outcomes in different contexts (Gil 1994). It is also difficult, with such approaches, to perform non-monotonic logical reasoning or merge new, unreliable information with existing beliefs. Research in logics has provided non-monotonic logical reasoning formalisms, e.g., ASP has been used in cognitive robotics (Erdem and Patoglu 2012). Researchers have combined ASP with inductive learning to monotonically learn causal laws (Otero 2003), and expanded the theory of actions to revise system descriptions (Balduccini 2007). Architectures have been developed to reason with hierarchical knowledge in first-order logic and process perceptual information probabilistically (Laird 2008). Many general frameworks have been developed that combine logical and probabilistic reasoning, e.g., Bayesian logic (Milch et al. 2006), first-order relational POMDPs (Juba 2016), and probabilistic extensions to ASP (Lee and Wang 2015). Algorithms based on classical first-order logic are often not expressive enough, e.g., modeling uncertainty by attaching probabilities to logic statements is not always meaningful. Algorithms based on logic programming tend not to support some of the desired capabilities such as efficient and incremental learning of knowledge, learning from interactions, and reasoning with large probabilistic components. Existing algorithms and architectures also do not support generalization as described in this paper.

Many formalizations have been proposed for representing, reasoning with, and learning affordances (Zech et al. 2017). Existing approaches represent affordances as possible effects of actions or behaviors (Guerin, Kruger, and Kraft 2013), or as emergent, functional and/or contextual properties based on attributes of the domain and the objects (Sarathy and Scheutz 2016). These approaches have used logics, probabilistic reasoning or a combination of both. Unlike these approaches, we build on research in psychology to formulate affordances as joint relations over attributes of one or more agents and objects in the context of specific actions (Langley, Sridharan, and Meadows 2018).

Interactive task learning is a general approach that includes learning concepts from domain observations and human demonstrations or instructions (Kirk, Mininger, and Laird 2016). It has often been posed as an RL problem, and relational RL (RRL) uses relational representations and regression for efficient Q-function generalization (Driessens and Ramon 2003; Tadepalli, Givan, and Driessens 2004). However, interactive relational learning algorithms typically limit generalization to a single planning task at a time, based on different function approximation or learning algorithms (Driessens and Ramon 2003; Bloch and Laird 2017), and do not support the commonsense reasoning capabilities desired in robotics. One exception was our prior work that combined ASP with RRL to discover some domain axioms and conditions under which specific actions cannot be executed (Sridharan, Meadows, and Gomez 2017; Sridharan and Meadows 2017). The architecture described in this paper combines the complementary strengths of declarative programming and relational learning through induction and reinforcement, for reasoning with and interactively revising incomplete domain knowledge.

## 5 Conclusions

This paper described an architecture for representing, reasoning with, and interactively learning actions' names, preconditions, effects, and objects over which they operate, along with associated affordances. Answer Set Prolog was used to represent and reason with incomplete domain knowledge for planning and diagnostics, and to guide interactive learning. The learning is achieved using decision-tree induction and relational reinforcement learning from observations obtained through active exploration, reactive action execution, and verbal descriptions from humans. Experimental results in a simulated domain indicate that our architecture supports reliable and efficient reasoning, and learning of actions and axioms corresponding to different types of knowledge. Inclusion of the learned actions and axioms in the system description improves the quality of the computed plans. In the future, we will explore the learning of actions and axioms in more complex domains and evaluate the architecture on physical robots, which will require the use of the component that reasons about perceptual uncertainty probabilistically. The long-term objective is to enable robots assisting humans to represent, reason with, and interactively revise different descriptions of incomplete domain knowledge.

## Acknowledgements

This work was supported in part by the Asian Office of Aerospace Research and Development award FA2386-16-1-4071, and the US Office of Naval Research Science of Autonomy award N00014-17-1-2434. All opinions and conclusions described in this paper are those of the authors.

## References

Balduccini, M., and Gelfond, M. 2003. Logic Programs with Consistency-Restoring Rules. In *AAAI Spring Symposium on Logical Formalization of Commonsense Reasoning*, 9–18.

Balduccini, M. 2007. Learning Action Descriptions with A-Prolog: Action Language C. In *AAAI Spring Symposium on Logical Formalizations of Commonsense Reasoning*.

Bloch, M. K., and Laird, J. E. 2017. Deciding to Specialize and Respecialize a Value Function for Relational Reinforce-

ment Learning. In *Multi-disciplinary Conference on Reinforcement Learning and Decision Making (RLDM)*.

Driessens, K., and Ramon, J. 2003. Relational Instance-Based Regression for Relational Reinforcement Learning. In *International Conference on Machine Learning*, 123–130. AAAI Press.

Erdem, E., and Patoglu, V. 2012. Applications of Action Languages to Cognitive Robotics. In *Correct Reasoning*. Springer-Verlag.

Gelfond, M., and Inclezan, D. 2013. Some Properties of System Descriptions of $\mathrm{AL_d}$. *Journal of Applied Non-Classical Logics, Special Issue on Equilibrium Logic and Answer Set Programming* 23(1-2):105–120.

Gil, Y. 1994. Learning by Experimentation: Incremental Refinement of Incomplete Planning Domains. In *International Conference on Machine Learning*, 87–95.

Guerin, F.; Kruger, N.; and Kraft, D. 2013. A Survey of the Ontogeny of Tool Use: from Sensorimotor Experience to Planning. *IEEE Transactions on Autonomous Mental Development* 5:18–45.

Juba, B. 2016. Integrated Common Sense Learning and Planning in POMDPs. *Journal of Machine Learning Research* 17(96):1–37.

Kirk, J.; Mininger, A.; and Laird, J. 2016. Learning Task Goals Interactively with Visual Demonstrations. *Biologically Inspired Cognitive Architectures* 18:1–8.

Laird, J. E. 2008. Extending the Soar Cognitive Architecture. In *International Conference on Artificial General Intelligence*.

Langley, P.; Sridharan, M.; and Meadows, B. 2018. Represention, Use, and Acquisition of Affordances in Cognitive Systems. In *AAAI Spring Symposium on Integrating Representation, Reasoning, Learning and Execution for Goal Directed Autonomy*.

Lee, J., and Wang, Y. 2015. A Probabilistic Extension of the Stable Model Semantics. In *AAAI Spring Symposium on Logical Formalizations of Commonsense Reasoning*.

Milch, B.; Marthi, B.; Russell, S.; Sontag, D.; Ong, D. L.; and Kolobov, A. 2006. BLOG: Probabilistic Models with Unknown Objects. In *Statistical Relational Learning*. MIT Press.

Miller, G. A. 1995. Wordnet: A lexical database for english. *Communications of the ACM* 38(11):39–41.

Otero, R. P. 2003. Induction of the Effects of Actions by Monotonic Methods. In *International Conference on Inductive Logic Programming*, 299–310.

Ramenzoni, V. C.; Davis, T. J.; Riley, M. A.; and Shockley, K. 2010. Perceiving Action Boundaries: Learning Effects in Perceiving Maximum Jumping-Reach Affordances. *Attention, Perception and Psychophysics* 72(4):1110–1119.

Sarathy, V., and Scheutz, M. 2016. A Logic-based Computational Framework for Inferring Cognitive Affordances. *IEEE Transactions on Cognitive and Developmental Systems* 8(3).

Someya, Y. 1998. e_lemma.txt (Version 2 for WordSmith 4).

Sridharan, M., and Meadows, B. 2017. A Combined Architecture for Discovering Affordances, Causal Laws, and Executability Conditions. In *International Conference on Advances in Cognitive Systems (ACS)*.

Sridharan, M.; Gelfond, M.; Zhang, S.; and Wyatt, J. 2017. A Refinement-Based Architecture for Knowledge Representation and Reasoning in Robotics. Technical report, `http://arxiv.org/abs/1508.03891`.

Sridharan, M.; Meadows, B.; and Gomez, R. 2017. What can I not do? Towards an Architecture for Reasoning about and Learning Affordances. In *International Conference on Automated Planning and Scheduling*.

Sutton, R. L., and Barto, A. G. 1998. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA.

Tadepalli, P.; Givan, R.; and Driessens, K. 2004. Relational Reinforcement Learning: An Overview. In *Relational Reinforcement Learning Workshop at International Conference on Machine Learning*.

Toutanova, K.; Klein, D.; Manning, C.; and Singer, Y. 2003. Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network. In *International Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 252–259.

Zech, P.; Haller, S.; Lakani, S. R.; Ridge, B.; Ugur, E.; and Piater, J. 2017. Computational Models of Affordance in Robotics: A Taxonomy and Systematic Classification. *Adaptive Behavior* 25(5):235–271.

# Integrated Planning and Execution for a Self-Reliant Mars Rover

**Steve Schaffer** and **Joseph Russino** and **Vincent Wong** and **Heather Justice** and **Daniel Gaines**
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, California 91109
*firstname.lastname*@jpl.nasa.gov

## Abstract

Planetary rovers exploring the surface of Mars face a challenging operational environment that requires close cooperation between deliberative planning and behavioral execution in order to most efficiently leverage the robot's capabilities into science value returned to earth. The Self-Reliant Rovers project envisions future rover missions that require only occasional high-level direction from human controllers to successfully conduct detailed in-situ studies of its Martian environs. To achieve this high degree of autonomy, this work leverages a spectrum of planning and execution techniques that allow the rover to respond appropriately to both opportunity and adversity it encounters. Small perturbations are accommodated at first by behavioral adaptation, with more and more extensive disruptions handled in turn by executive administration of plan flexibility, heuristic-guided plan repair strategies, and finally comprehensive replanning from science campaign goals. The integrated system has been deployed and tested on a terrestrial rover in an environment and under scenarios that anticipate those faced by future Mars rovers. This paper recounts complexities of planning and execution coordination faced in the rover domain and the practical solutions employed to address them. Particular emphasis is given to lessons from the field and foibles ripe for remedy by future advances in planning and execution research.

## Introduction

Current planetary rover operations for Mars Science Laboratory (Vasavada et al. 2014) exert a significant daily workload on mission operations staff. Within one work shift, scientists and engineers must interpret downlinked data, reevaluate overarching mission goals, and then synthesize a responsive activity plan and detailed command sequence covering the next planning period. The nominal single martian day planning period allows the team to maintain a high level of productivity via timely manual response to execution eventualities, but there is markedly diminished efficiency when plans must span multiple days (Gaines et al. 2017a) (e.g. due to off-sync relay passes or institutional holidays.) Future Mars surface missions are anticipated to have even more frequent and extensive lapses in regular communication due to turnover in available relay craft (Edwards et al. 2014), and will thus require more robustly integrated onboard planning and execution to sustain productivity absent prompt human feedback. Even with regular communication, the productivity of current rover missions has benefited greatly from onboard collaboration of imaging activity execution with automated data analysis and goal selection (Francis et al. 2017).

The Self-Reliant Rovers architecture seeks to further extend onboard planning and executive capabilities so that future rover missions can continue productive scientific inquiry without constant micromanagement by human controllers. The SRR architecture was borne out of thorough study of current rover operations and the potential efficiencies attainable by increased onboard autonomy (Gaines et al. 2016). Under the SRR architecture, rover objectives can be expressed as high-level campaign intents rather than meticulously assembled daily activity plans (Gaines et al. 2017b). For example, a scientist might request detailed imagery of any quartz veins detected during a walk about of a boulder field, while another may request recurring atmospheric opacity measurements every day at noon. Engineers might specify mandatory relay communication passes along with required battery reserves at the end of the planning period. The onboard planning and executive functions can leverage the discretion entailed in such conceptual guidance to respond directly to unpredictable outcomes and continue exploration during communication gaps.

This paper outlines challenges to effectively integrated planning and execution within the Mars rover domain, along with practical techniques employed within and between SRR components to achieve the envisioned level of rover mission autonomy. Initial results on a terrestrial rover test bed are presented, and fertile areas for future enhancements to the integrated system are described.

## Approach

The Self-Reliant Rover system is designed within the context of the Jet Propulsion Laboratory flight software architecture (Weiss 2013) and incorporates a tiered robotic control architecture. At the highest level, scientists and engineers use the MSLICE graphical interface (Powell et al. 2009) to construct both general campaign objectives and detailed constraints that will guide the system's behavior. The goals are then transmitted to an onboard optimizing activity planner, CASPER (Chien et al. 2000), which assembles and maintains a comprehensive working schedule for

Figure 1: The Athena rover imaging a science target during a simulated Mars surface mission.

the rover that fulfills the requests while respecting vehicle safety limits. Activities are then dispatched from this schedule to a purpose-built reinterpretation of the MEXEC state-based executive (Verma et al. 2017) that oversees their coordinated execution and reports back on their ongoing status. The executive calls upon task-oriented behavior components, and those in turn refer to low-level functional components, that together accomplish specific robotic tasks such as locomotion, imagery, and data analysis. Many components are reused from the CLARAty library of portable robotic software modules (Volpe et al. 2001). Channelized robot state and resource updates are published by the cognizant components, and may be subscribed to by any other interested component. Component encapsulation and intercommunication are provided by the ROS framework (Quigley et al. 2009).

**Planning** Updated high-level objectives are provided to the onboard planner by the ground operations team on an intermittent schedule, but special onboard system nodes are also empowered to submit new constraints and goals to the planner. Using guidance from scientists, the automated science data analysis component recognizes features in acquired imagery that may warrant further study, such as boundaries between geologic deposits. The analysis component calculates the location of the new targets in the environment using camera model and rover position metadata that is attached to each image, and constitutes new goals for follow-up science activities based on a template provided by the science team. The template allows humans to bound the self-directed goal behavior by specifying details such maximum number of follow-ups, total priority/utility to assign, and association to broader science campaigns. The new goals are submitted to the planner and are integrated into future decisions along with the rest of the pending goals and constraints. Similarly, a vehicle health management component monitors ongoing rover performance and may respond to anomalous trends by imposing tighter safe operation limits or calling for diagnostic activities. The planner remains the arbiter of when to undertake new activities since it has a broader picture of the rover's future plan, including upcoming critical activities such as communication passes.

A key feature of the system is the onboard planner's flexibility to either heuristically repair the existing plan in the face of small perturbations or to regenerate a new plan forward from the as-executed stem in the case of more extensive disruptions. Replanning is not especially prohibitive ($<$1 minute), thanks to the efficient domain-specific multi-threaded best-first branch-and-bound anytime path/plan optimization algorithm employed, but it is still intensive enough that full replanning cannot be performed every update cycle. Accordingly, the criteria under which full replanning is invoked are relatively liberally construed as those which have reasonable probability of meaningfully changing the the path among goal locations or the science and engineering activities conducted along the route. A incoming batch of new goals thus always triggers replanning, as does failure of an activity declared by the executive. Replanning is also invoked when updates to rover state or resource levels propagate into predicted conflicts in the future plan, for example due to a late-running motor preheating task. A more subtle trigger examines the unused resource and time margins, and calls for replanning if the excess overtakes some threshold, as might be the case after a series of better-than-expected executions.

Because they are less likely to result in structural changes to the plan, minor updates that reach the planner are handled by rapid plan repair heuristics. Resource and state updates are posted to the plan at the time they are received, with fast re-prediction propagating the timeline's expected future values and checking for any conflicts with upcoming activities or constraints. Importantly, timeline updates that do not immediately trigger predicted conflicts are still collected and posted to the plan to inform future predictions, which may finally rise to a conflict only after several small discrepancies are recorded. When activities end early, a dynamic packing heuristic adjusts future action start times as close to the present as avoids inducing any plan conflicts. Any associated activities, such as mechanism preheating, are also moved forward. This results in filling unused blocks of time following hastened activities, while leaving absolute-timed activities such as communication passes at their proper time.

Late running actions cannot be accommodated in the same manner in general; by the time a preceding action is known to be late, the subsequent activity may have already been dispatched to the executive. The main loop of the planner must dispatch activities well enough in advance so that no start times are missed during its full iteration duration, which may be extensive in cases where full replanning is invoked. The current SRR system inherits a fixed duration commit window that the anytime path solver algorithm is obliged to respect: activities within the window must not be modified since they were already dispatched to the executive, and the solver must return control to the main loop by the end of the window, regardless of its solution progress. As an online algorithm, the solver is able to submit the best plan so far at any break point, and is also able to restart from its previous partial solution state. Deconflicting dispatched activities when one runs past its planned end time becomes

the responsibility of the executive, but is informed by the planner's model of the individual activity state and resource constraints, which are also passed down.

**Execution** When the designated start time for a dispatched action arrives, the executive first checks the constraints from the planner before initiating the activity. If all of the constraints are not met (for example, a late running panorama is still using the rover mast also needed for a targeted image), the subsequent activity start is held back by the executive until either the constraints are satisfied or a delay threshold is reached. The planner may select different delay thresholds for each dispatched activity instance in order to communicate contextual start time flexibility from the plan constraints. The eventual activity start and end times are reported by the executive to the planner and other interested components to ensure accurate resource modeling and vehicle health assessment. This approach allows the executive to locally handle small delays that do not have a large impact on the plan structure, but in a way that is consistent with the planner's expectations of activity preconditions.

The executive contains another layer of precondition checking for safety purposes: a hard-coded table from the system engineering team that delineates which activities are safe to execute concurrently with each other and specific rover states. For example the drive action might require that the rover arm is in the stowed state and that no other activity is using the navigation cameras. Before finally initiating any new activity that is otherwise cleared for execution, the executive checks it and any other ongoing activities against this compatibility table. A failure to pass the safety check results in rejection of the new activity by the executive and failure notification being sent to the planner. This is different than for planner supplied constraints, which merely delay execution in anticipation of imminent state updates. A safety check rejection indicates that the planner is direly unaware of the current execution conditions, or fails to model an important system safety rule.

After the executive initiates an activity, it makes calls to the required lower-level behavioral task components and then continues monitoring their ongoing progress with a small state machine. Simple tasks just return a status message on their completion, in which case the executive notes the end time of the activity, clears the state machine, and forwards the result (success or failure) to the planner. More sophisticated activities include additional monitoring of behavior start up and progress reports, enforcement of rover state conditions throughout the activity execution (akin to the precondition safety checks), and activity termination criteria monitoring. In the event a monitored in-condition of the activity is violated, the behavior is signaled to immediately abort in order to prevent vehicle damage, and the planner is notified of the activity's failure. By default, activities are also checked against their planned end-time; any overruns past a chosen threshold trigger the same executive abort response. This means that the planner model of action durations must be pessimistically long, though the resultant plan inefficiencies are largely recovered during execution by the dynamic packing and replanning features of the overall system.

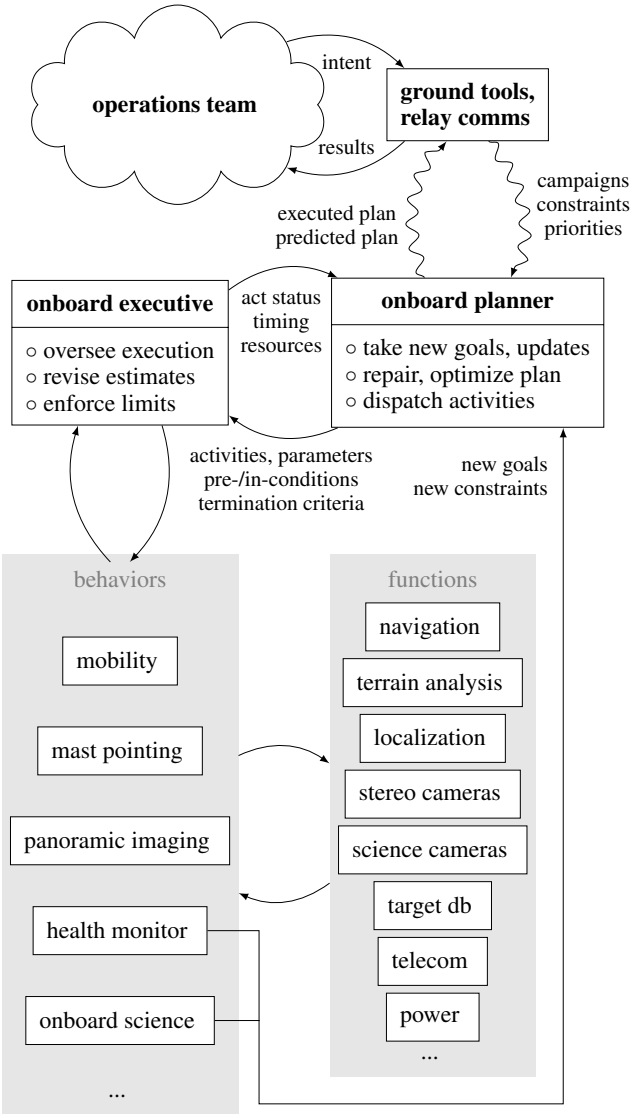



Figure 2: Self-Reliant Rover architecture overview. Scientists and engineers provide high level objectives, which are optimized in-situ by an onboard planner. The resultant activities are managed by an executive that understand constraints and flexibility in the plan. Individual behaviors invoke even lower-level modular functions. The executive and planner cooperate to respond to unexpected outcomes, changing resource estimates, and even new goals and constraints borne out through execution.

**Driving** The drive behavior component includes many additional features to allow close cooperation between the planner and executive in meeting the challenges posed by the rugged Martian terrain. Drive activities serve as connectors between all of the science activity locations in the plan, but are highly complex planning endeavors in their own right, as they continuously vary the rover's position and resource use over an extended period. Plausible science or engineering campaigns requested to recur e.g. every 50 meters of

drive distance, every 10 meters of elevation change, or every 3 hours of elapsed time are all profoundly impacted by the precise drives in a plan. Further interactions are mediated by the stationary state requirement of some activities such as fixed communication passes, regenerative sleeping, and almost all science activities. On top of this, drives represent the most significant execution uncertainty for planetary rovers due their highly-coupled interactions with the unexplored alien environment.

Overview orbital imagery can only give rough clues about the surfaces and obstacles that will be encountered along different possible drive routes. Rovers have thus long relied on onboard stereo vision, obstacle avoidance, and visual odometry embedded in the locomotion services (Goldberg, Maimone, and Matthies 2002), and SRR continues this tradition. The unpredictable diversions around obstacles can lead to significant discrepancies with estimated arrival times, as well as drawing the rover off of its expected path. The primary impact of driving delays is on the scheduling of subsequent activities, either because they required the rover be at a specific location or to be stationary, but there are also downstream effects on rover resource and state predictions as well. In case of diversions significantly off of the planned course, it may also become appropriate to adjust the plan to accomplish other goals along the detour route before resuming the initial drive. In extreme cases, the system may have to accept the eventuality that a selected target destination is really unreachable.

The constraint-based task execution strategy discussed above can accommodate some of the drive dependencies, for example by providing a location precondition on targeted science activities. This raises a question about how physical rover positions correspond to locations specified in a campaign request, which is answered jointly by the locomotion engine itself and a target association component. When determining location satisfaction, the associator takes into account additional details from the campaign such as permissible instrument range to intended target, allowed rover bearings (e.g. for illumination reasons), and instrument field of view limits. The locomotion engine has been augmented to accept the arguments when requesting a drive as well, whether those drives are for immediate execution or for hypothetical evaluation during goal planning. Using the same locomotion engine at plan and execution time ensures that the planner benefits from the latest obstacle maps and thus has a more accurate prediction of the drive behavior.

In addition to unpredictable obstacle environments, drive actions may be subject to further uncertainty due to the driving surface. The substrate texture can change abruptly from hard rock outcrop, to compacted soil, to loose sandy ridges, each with very different wheel slip characteristics. Furthermore, the slope of the terrain and the rover's approach aspect also impact drivability. Certain combinations of terrain features even amount to mission-ending wheel trap hazards that must be fastidiously avoided. Terrain classification and slip aware navigation components were incorporated into SRR to address these uncertainties (Rothrock et al. 2016). Similar to obstacle avoidance, imagery collected during drives is used to classify different textured soil types and slope inclines around the rover into different cost categories. The costs are overlaid onto the constantly updating navigation map used by the locomotion engine so that it can update its routes to avoid dangers and make the fastest progress to the goal.

**Drive Termination** When the locomotor updates its route during execution, it diverges from the time and distance estimates predicted at plan time. The locomotor posts progress reports that include revised estimates, which are then used by the executive to predict the likelihood of success or failure by the planned end time. Rather than waiting until that end time to post a failure to reach a target location, the executive is empowered to abort the drive early when revised estimates exceed the allotted time frame by some margin. This minimizes wasted driving effort in difficult terrain by allowing more immediate replanning with new map data. It also indirectly triggers replanning on large diversions from the expected route, since such diversions are likely to induce large changes in duration estimates.

The executive is also able to terminate drive actions early in order to meet campaign objectives based on rover states such as distance driven or time of day. This is necessary since an initially planned drive action may end up driving further or taking longer than expected, so much so that the next instance of a recurring campaign activity should be invoked. For example, if the planner must accommodate an image request every 50 meters driven, then an upward revision from 40 to 60 meters estimated drive distance requires stopping for an extra intermediate image. To accommodate this scenario, the planner looks ahead of each drive it dispatches to find recurring campaign goals may need to intercede in the drive, and attaches those campaign criteria to the drive as additional termination conditions. The executive monitors these termination conditions (e.g. odometer reading of 50 meters, or specific time of day) and aborts the drive behavior when they are met. Rather than reporting outright failure of the drive, the executive reports which termination condition stopped the drive and the actual location different from intended target location. The planner nevertheless interprets the unexpected stop as an inconsistency in its current plan, and so invokes replanning, which will likely insert the relevant campaign activity followed by a completion of the initial drive. The rover states to which such termination criteria are attached should be carefully selected to avoid ambiguities due to partial drives; for example specific odometer or clock readings should be used rather than distance or time driven from the start of a drive segment. This allows the criteria to be applied uniformly across drive segments rather than recomputed relative to each leg.

## Results

The SRR system was demonstrated on the JPL Athena rover within a mission scenario that explores the JPL mini-Mars Yard robotic testing environment. The primary science objective was to characterize the rock outcrop materials embedded in the sandy soil using the rover's mast-mounted cameras. The simulated mission spans a period of limited communication with operators, so the rover must operate al-

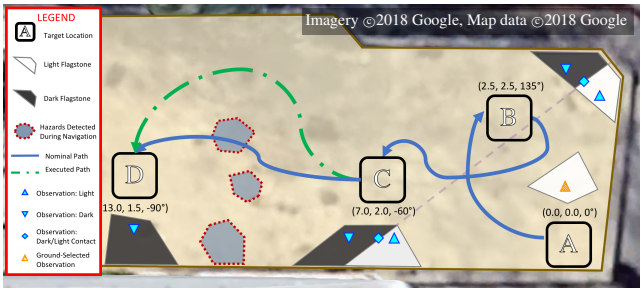



Figure 3: Overview of simulated mission area. Operator inputs include a specific target selection (orange) near starting area A along with only high-level campaign guidance for areas B, C, and D. Automated science analysis injects additional targets (cyan) during execution. The initial planned route (blue) is dynamically adjusted (green) to avoid unanticipated terrain hazards (red).

most entirely autonomously in order to remain productive toward its high-level goals.

Figure 3 shows the overhead layout of the mission area, as might be available to mission planners from orbital imagery. The operations team selects several regions of interest (indicated by letters) from this coarse data, but is unable to identify specific targets or terrain obstacles beyond a few meters from the rover. They then construct a goal for each target area that entails driving to a specified vantage point, acquiring a contextual wide-angle image, and then running the automated science algorithms. The planner will stitches these goals together in an optimal drive ordering that achieves as many as possible. The scientists also create campaign goals for the desired follow-up outcrop observations in each area, including templates for goals automatically generated by the onboard science analysis. The planner and automated science cooperate to identify the best candidate targets to include in the plan so as to maximize expected utility score. In this demonstration scenario, campaigns request follow-up mast camera imaging of the 2-5 best outcrop specimens in each category at each location. Several additional relevant campaign types were demonstrated in separate scenarios. For example, the operators can specify ongoing temporal periodic campaigns such as visual atmospheric opacity ($\tau$) measurements every $20\pm2$ minutes. Mandatory downlink relay communication passes can also be enforced at specific times in the schedule, representing a exogenous orbiter overflights.

All of the various goals are provided to the rover at its morning communication pass at the start of the mission scenario. Thereupon, the onboard planner generates a plan to image the specifically requested target near A, and then travel in turn to B, C, and D to conduct survey observations (fig.4, top, and fig.3, blue path). The plan adheres to all rover resource limits (such as battery energy and data volume), as well as incorporating any required heating (such as needed for instruments or mobility mechanisms). The actual path driven by the rover undergoes refinement by the onboard terrain classification and autonomous navigation so as

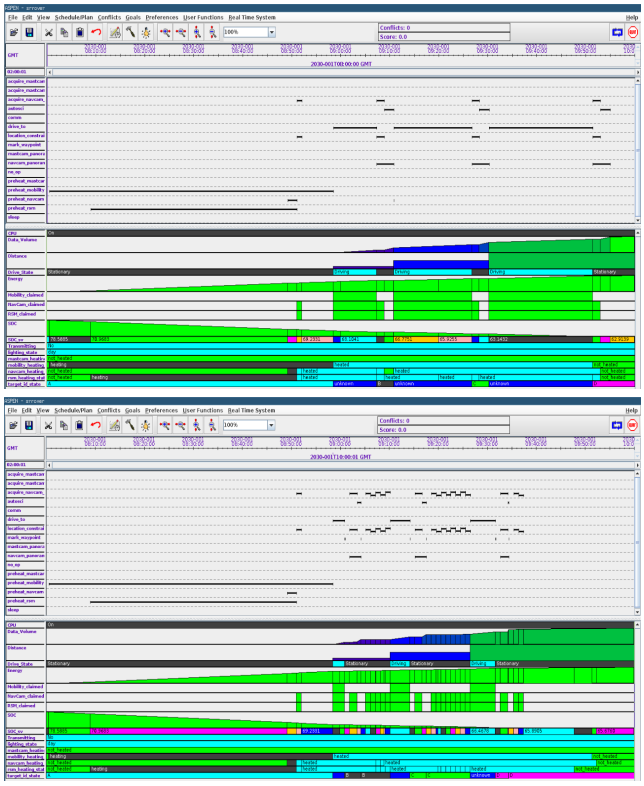


Figure 4: Initial generated plan and final as-executed plan for the simulated mission scenario. Many new targeted science goals are suggested at run-time by automated image analysis and then integrated into the schedule in service of science campaigns. Drive estimates are also updated during execution, thus correcting initial approximations.

to best avoid obstacles along the planned route (fig.3, green path). Diversion delays and expeditious travel cause minor perturbations to the plan, which are accommodated by the dynamic packing plan heuristic.

On arriving at B, and later C, the rover acquires the requested contextual images and analyzes them using the onboard science detectors, which in turn identify flagstone outcrops for follow-up imaging. The selected targets are then automatically injected as new goals into the planner campaigns, and a replanning cycle is initiated. The planner's updated solution includes each of the newly suggested observations, which are duly collected before proceeding to the next area.

Upon driving toward D, the rover's automated terrain classification identifies a major obstacle, and the navigation system must divert significantly. The planner incorporates updated drive estimates from the navigation engine to ensure that the plan can accommodate the delay without conflict. After planning a safe path around the observed obstacles and eventually reaching D, the system once again identifies flagstone features and conducts the requested follow-up observation. At this point the simulated mission ends.

As seen in the final plan (fig.4, bottom), the productiv-

ity benefits of additional onboard rover autonomy are evident even within the limited scope of this demonstration scenario. Traditional operations would have accomplished just one initial outcrop observation and a first drive. The combined autonomy of the SRR system produced three survey panorama images throughout the mission area, toured several unexpectedly difficult terrain routes, and accrued fifteen additional targeted outcrop observations. The scenario successfully demonstrated the mission productivity benefits of integrated planning and execution within the SRR architecture.

## Future Work

A consistent decision framework for when each class of plan modification is warranted would help theoretically ground the ad hoc assignment of replanning versus repair triggers. Such a framework would likely consider a balance between the maximum or estimated utility payoff of invoking replanning versus its opportunity cost, measured both in terms of computation time as well as operational costs of disrupting an existing plan that human operators may have a stake in. Useful clues to the potential payoff are likely available as an ancillary product of the planner's branch-and-bound algorithm. New goals with large expected rewards would thus drive replanning when they became available, while several smaller adjustments would have to accumulate before warranting a replan.

The current system operates without reserving resources or time for follow-up science goals anticipated for each automated science analysis run, instead depending on agile replanning to fit in the new goals when they appear. Other possible approaches where only given cursory evaluation and deserve further study. One alternate approach is to use a placeholder activity that represents the budget of time and resources that are allotted to automatic science goals, and then decrement that reserve for each actually planned follow-up. The reserve could be over the entire planning period or attached to each separate automated science activity. Planning for the automated science analysis follow up observations also faces a classic dilemma of exploitation versus exploration. Since new nearby goals are injected following the initial image analysis, there is a tendency for the rover to get caught up in examining that first location and defer subsequent targets that could have even more valuable follow ups. Budgeting time and resources for each analysis target will help reduce the early over-exploitation, but really solving the problem requires proper incorporation of some exploration metric into the plan scoring heuristic itself. Such a metric is counter-intuitive to normal path planning since it requires driving the entire walkabout distance first to do initial analysis, and then driving it again to revisit the most interesting targets.

Tighter collaboration on activity duration adjustment is possible between the planner and executive. This would reduce the need to consistently overestimate activity durations within the planner model and the associated inefficiencies, as well as reducing the occurrence of executive delay holds on subsequent activities committed early. The planner could evaluate the range of conflict-free durations for each dispatched activity and pass along those bounds as a end time flexibility, rather than enforcing the single predicted end time as a hard limit on execution. The executive could manage the flexibility to extend activities that need just a little more time for completion without worry of damaging the plan. Alternately, the executive could pose requests for activity extensions to the planner as soon as it received revised estimates from the behaviors. The planner could then perform a hypothetical planning cycle with the extended activity and report back to the executive whether the extension is granted.

Additional integration between the locomotion components and the planner would also benefit the system. In particular, the terrain classification and obstacle detection systems are constantly improving their map of the environment, but the planner only benefits from those improvements when full replanning is invoked. Until then, the planner relies on its previously cached drive estimate responses. Instead, it may be worthwhile for the locomotion components to track which estimates the planner is currently using, and to transmit revisions to those estimates when optimal routes change by some threshold. The planner also only indirectly recognizes when the driven route has diverged significantly from the initial path via changing drive time estimates. This means that the planner is slow to respond to opportunities nearby the diverted route, perhaps even missing them completely. If the planner, executive, and locomotor collaborated on the actual path geometry (rather than just summary estimates), the system would be able to capture such detour opportunities.

Many integration hurdles could be overcome if the planner and executive shared access to the same plan. For example, the serialization of planner constraints and flexibility could be avoided, as could the convolutions of advance dispatch. However, it would require great care to ensure that ongoing execution updates could successfully interleave with the planner during active replanning. Even without such drastic merging, the commit window scheme could be improved to increase planning flexibility and reduce reliance on executive delay holds. For example, the commit window could be dynamically sized depending on execution conditions: narrow when only minor changes are being posted, but wider when full replanning is called for.

There are several interesting issues to tackle regarding integration of mission planning on Earth and with the onboard planning and execution. Because of the communication delays involved, the human planners are always operating with an old snapshot of what the rover had accomplished so far at the time of downlink. The onboard planner also communicates its then-current plan for the rest of the period so that human operators have a concept of what goals might be achieved by the time their new requests would arrive. However, the onboard planner may diverge from that plan for various reasons, meaning operators must not rely on any specific future chain of events. A balance may be struck by assigning probabilities to possible futures and expressing new goals either independent of unconfirmed actions or explicitly conditional on them. When the rover receives goals

updates during an uplink pass, it must also carefully disposition each change within the actually executed context but with reference to the knowledge state of the humans when they formulated the requests. For example, operators may call for removal of a goal that was actually already accomplished, or they may request a loosely targeted goal whose precise location has since been more accurately determined by the rover.

Keeping a consistent vehicle model synchronized among all the components of SRR is an outstanding challenge. While a single activity dictionary serves as the original source for the planner and executive models, regeneration from the source is only automated for the planner model, and even then involves some additional manual tweaks. The locomotion components are fully independent and must be kept in sync manually, e.g. when the rover speed is updated. Fully automated generation of each component's model from a single spacecraft description would be better. This would also allow more direct correspondence between planner-level abstractions with the underlying vehicle states reported by low-level components.

## Conclusion

Effective integration of planning and execution components within the Self-Reliant Rover architecture enables the robotic explorer to operate productively for long periods with only high-level guidance from human operators. The rover domain presents many unique demands on such an integrated system, which have been addressed by a range of practical techniques, with varying degrees of complication and success. The system was deployed on the Athena rover and demonstrated within a simulated Mars mission vignette, where it realized significant productivity gains over traditional operations techniques. Despite this achievement, there are still many open avenues for tighter integration among the system's planning and execution components.

## Acknowledgments

Portions of this work were performed at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

## References

Chien, S.; Knight, R.; Stechert, A.; Sherwood, R.; and Rabideau, G. 2000. Using iterative repair to improve responsiveness of planning and scheduling. In *International Conference on Artificial Intelligence Planning Systems (AIPS 2000)*.

Edwards, C. D.; Barela, P. R.; Gladden, R. E.; Lee, C. H.; and Paula, R. D. 2014. Replenishing the mars relay network. In *2014 IEEE Aerospace Conference*, 1–13.

Francis, R.; Estlin, T.; Doran, G.; Johnstone, S.; Gaines, D.; Verma, V.; Burl, M.; Frydenvang, J.; Montaño, S.; Wiens, R. C.; Schaffer, S.; Gasnault, O.; DeFlores, L.; Blaney, D.; and Bornstein, B. 2017. Aegis autonomous targeting for chemcam on mars science laboratory: Deployment and results of initial science team use. *Science Robotics* 2(7).

Gaines, D.; Doran, G.; Justice, H.; Rabideau, G.; Schaffer, S.; Verma, V.; Wagstaff, K.; Vasavada, A.; Huffman, W.; Anderson, R.; Mackey, R.; and Estlin, T. 2016. Productivity challenges for Mars rover operations: A case study of Mars Science Laboratory operations. Technical Report D-97908, Jet Propulsion Laboratory.

Gaines, D.; Doran, G.; Justice, H.; Rabideau, G.; Schaffer, S.; Verma, V.; Wagstaff, K.; Vasavada, A.; Huffman, W.; Anderson, R.; Mackey, R.; and Estlin, T. 2017a. A case study of productivity challenges in mars science laboratory operations. In *International Workshop on Planning and Scheduling for Space (IWPSS 2017)*.

Gaines, D.; Rabideau, G.; Doran, G.; Schaffer, S.; Wong, V.; Vasavada, A.; and Anderson, R. 2017b. Expressing campaign intent to increase productivity of planetary exploration rovers. In *International Workshop on Planning and Scheduling for Space (IWPSS 2017)*.

Goldberg, S. B.; Maimone, M. W.; and Matthies, L. 2002. Stereo vision and rover navigation software for planetary exploration. In *Aerospace Conference Proceedings, 2002. IEEE*, volume 5, 5–5. IEEE.

Powell, M. W.; Shams, K. S.; Wallick, M. N.; Norris, J. S.; Joswig, J. C.; Crockett, T. M.; Fox, J. M.; Torres, R. J.; Kurien, J. A.; and McCurdy, M. P. 2009. Mslice science activity planner for the mars science laboratory mission. Technical report, Jet Propulsion Laboratory.

Quigley, M.; Conley, K.; Gerkey, B.; Faust, J.; Foote, T.; Leibs, J.; Wheeler, R.; and Ng, A. Y. 2009. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, 5.

Rothrock, B.; Kennedy, R.; Cunningham, C.; Papon, J.; Heverly, M.; and Ono, M. 2016. Spoc: Deep learning-based terrain classification for mars rover missions. In *AIAA SPACE 2016*, 5539.

Vasavada, A.; Grotzinger, J.; Arvidson, R.; Calef, F.; Crisp, J.; Gupta, S.; Hurowitz, J.; Mangold, N.; Maurice, S.; Schmidt, M.; et al. 2014. Overview of the mars science laboratory mission: Bradbury landing to yellowknife bay and beyond. *Journal of Geophysical Research: Planets* 119(6):1134–1161.

Verma, V.; Gaines, D.; Rabideau, G.; Schaffer, S.; and Joshi, R. 2017. Autonomous science restart for the planned europa mission with lightweight planning and execution. In *International Workshop on Planning and Scheduling for Space (IWPSS 2017)*.

Volpe, R.; Nesnas, I.; Estlin, T.; Mutz, D.; Petras, R.; and Das, H. 2001. The claraty architecture for robotic autonomy. In *Aerospace Conference, 2001, IEEE Proceedings.*, volume 1, 1–121. IEEE.

Weiss, K. 2013. An introduction to the jpl flight software product line. In *Proceedings of the 2013 Workshop on Spacecraft Flight Software (FSW-13)*.