29th International Conference on
Automated Planning and Scheduling

July 11-15, 2019, Berkeley, CA, USA

IntEx 2019

Proceedings of the 3rd Workshop on

Integrated Planning, Acting, and Execution

Edited by:

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

Organization

Mark Roberts, Naval Research Laboratory, USA

Tiago Vaquero, Jet Propulsion Laboratory, USA

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

Alessandro Cimatti, FBK

Jeremy Frank, NASA Ames, USA

Dan Magazzeni, Kings College London, UK

Fabio Mercorio, University of MilanBicocca, Italy

Reuth Mirsky, Ben Gurion University

Bob Morris, NASA Ames, USA

Vikas Shivashankar, Amazon Robotics, USA

Roni Stern, Ben Gurion University, Israel

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. 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). 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, bringing to the foreground concerns over how much computational effort planning should invest over time. But reality rarely proceeds according to the plan or the model. Planning, plan execution, diagnosis, and causal explanation have each been examined by various research efforts, but discussion of the linkages between them in the literature is still somewhat sparse. When considering how to integrate these functions, at least three questions must be considered: (1) System integration: how to integrate planning, plan execution, diagnosis, and causal explanation in a single system? (2) Model / Belief updates: when the unexpected happens, how does the system change its internal representation so future plans are effective? (3) Replanning: what to do now that the unexpected has happened?

Performing diagnosis while executing a plan gives a planning and execution agent an opportunity to recover from failures; however, it also raises many new issues. These include sharing reasoning time between planning and diagnosis, trading off execution resources between goal achievement and diagnostic testing (for active diagnosis), and how to act in the presence of multiple competing diagnoses.

The Third IntEx workshop aims to provide a forum for discussing the challenges of integrating planning with execution, emphasize the role of diagnosis in online planning and execution, and raise awareness, promote discussion, and encourage cross-fertilization of ideas from the following topics:

- Integration of planning and plan execution
  - Theory (e.g. flexibility vs uncertainty, replanning vs contingent planning algorithms)
  - Practice (technologies, architectures, system integration)
- Integration of planning and fault management (diagnosis, prognostics, anomaly detection) technologies:
  - Planning to Diagnose (active diagnosis)
  - Planning and Fault Model Integration (impact of diagnosis algorithms on plan model revision, level of abstraction of models)
- Integration of planning and causal explanation (state/event estimation and prediction) technologies:
  - Improving or revising plans based on inferred causal explanations
  - Revising long-term models based on causal explanation

In addition to the above special focus areas, we encourage short or long papers on past topics of interest to this workshop including: benchmarks or challenge problems for integrated execution; improving planning performance from execution experience; plan dispatching or plan executives; anytime or incremental planning; execution monitoring, comparing replanning, plan repair, regoaling, or plan merging; managing open worlds with closed-world planners; model learning from experience or determining an observation policy; policy switching or applying incremental policy adjustment.

Mark Roberts, Tiago Vaquero, Tim Niemueller, and Simone Fratini
June 2019

Contents

Enabling Limited Resource-Bounded Disjunction in Scheduling
Jagriti Agrawal, Wayne Chi, Steve Chien, Gregg Rabideau, Stephen Kuhn and Daniel Gaines **1**

On Expected Value Strong Controllability
Jeremy Frank **10**

Dynamic Controllability with Single and Multiple Indirect Observations
Paul Morris and Arthur Bit-Monnot **19**

Executing Contingent Plans: Addressing Challenges in Deploying Artificial Agents
Christian Muise, Miroslav Vodolan, Shubham Agarwal, Ondrej Bajgar, Luis Lastras and Josef Ondrej **28**

A Hybrid Planning and Execution Approach Through HTN and MCTS
Xenija Neufeld, Sanaz Mostaghim and Diego Perez-Liebana **37**

Interleaving Acting and Planning Using Operational Models
Sunandita Patra, Malik Ghallab, Dana Nau and Paolo Traverso **46**

Executing Multi-Goal Mission Plans for Coordinated Mobile Robots
Marlyse Reeves, Enrique Fernandez Gonzalez and Brian Williams **55**

Monitoring Numeric Expectations in Goal Reasoing Agents
Noah Reifsnyder and Hector Munoz-Avila **63**

Automated Verification of Social Laws Robustness for Reactive Agents
Alexander Tuisov and Erez Karpas **71**

Enabling Limited Resource-Bounded Disjunction in Scheduling

Jagriti Agrawal, Wayne Chi, Steve Chien, Gregg Rabideau, Stephen Kuhn, and Dan Gaines
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109
{firstname.lastname}@jpl.nasa.gov

Abstract

We describe three approaches to enabling an extremely computationally limited embedded scheduler to consider a small number of alternative activities based on resource availability. We consider the case where the scheduler is so computationally limited that it cannot backtrack search. The first two approaches precompile resource checks (called guards) that only enable selection of a preferred alternative activity if sufficient resources are estimated to be available to schedule the remaining activities. The final approach mimics backtracking by invoking the scheduler multiple times with the alternative activities. We present an evaluation of these techniques on mission scenarios (called sol types) from NASA's next planetary rover where these techniques are being evaluated for inclusion in an onboard scheduler.

Introduction

Embedded schedulers must often operate with very limited computational resources. Due to such limitations, it is not always feasible to develop a scheduler with a backtracking search algorithm. This makes it challenging to perform even simple schedule optimization when doing so may use resources needed for yet unscheduled activities.

In this paper, we present three algorithms to enable such a scheduler to consider a very limited type of preferred activity while still scheduling all required (hereafter called *mandatory*) activities. Preferred activities are grouped into *switch groups*, sets of activities, where each activity in the set is called a *switch case*, and exactly one of the activities in the set must be scheduled. They differ only by how much time, energy, and data volume they consume and the goal is for the scheduler to schedule the most desirable activity (coincidentally the most resource consuming activity) without sacrificing any other mandatory activity.

The target scheduler is a non-backtracking scheduler to be onboard the NASA Mars 2020 planetary rover (Rabideau and Benowitz 2017) that schedules in priority first order and never removes or moves an activity after it is placed during a single run of the scheduler. Because the scheduler does not backtrack, it is challenging to ensure that scheduling a consumptive switch case will not use too many resources and therefore prevent a later (in terms of scheduling order, not necessarily time order) mandatory activity from being scheduled.

The onboard scheduler is designed to make the rover more robust to run-time variations by rescheduling multiple times during execution (Gaines et al. 2016a). If an activity ends earlier or later than expected, then rescheduling will allow the scheduler to consider changes in resource consumption and reschedule accordingly. Our algorithms to schedule switch groups must also be robust to varying execution durations and rescheduling.

We have developed several approaches to handle scheduling switch groups. The first two, called guards, involve reserving enough sensitive resources (time, energy, data volume) to ensure all later required activities can be scheduled. The third approach emulates backtracking under certain conditions by reinvoking the scheduler multiple times. These three techniques are currently being considered for implementation in the Mars 2020 onboard scheduler.

Problem Definition

For the scheduling problem we adopt the definitions in (Rabideau and Benowitz 2017). The scheduler is given

- a list of activities
  $A_1\langle p_1, d_1, R_1, e_1, dv_1, \Gamma_1, T_1, D_1\rangle \ldots$
  $A_n\langle p_n, d_n, R_n, e_n, dv_n, \Gamma_n, T_n, D_n\rangle$
- where p_i is the scheduling priority of activity A_i;
- d_i is the nominal, or predicted, duration of activity A_i;
- R_i is the set of unit resources $R_{i_1} \ldots R_{i_m}$ that activity A_i will use;
- e_i and dv_i are the rates at which the consumable resources energy and data volume respectively are consumed by activity A_i;
- $\Gamma_{i_1} \ldots \Gamma_{i_r}$ are non-depletable resources used such as sequence engines available or peak power for activity A_i;
- T_i is a set of start time windows $[T_{i_{j\_start}}, T_{i_{j\_preferred}}, T_{i_{j\_end}}] \ldots [T_{i_{k\_start}}, T_{i_{k\_preferred}}, T_{i_{k\_end}}]$ for activity A_i. [1] ;

[1] If a preferred start time, $T_{i_{j\_preferred}}$ is not specified for window j then it is by default $T_{i_{j\_start}}$

- D_i is a set of activity dependency constraints for activity A_i where $A_p \rightarrow A_q$ means A_q must execute successfully before A_p starts.

The goal of the scheduler is to schedule all mandatory activities and the best switch cases possible while respecting individual and plan-wide constraints.

Each activity is assigned a *scheduling priority*. This priority determines the order in which the activity will be considered for addition to the schedule. The scheduler attempts to schedule the activities in priority order, therefore: (1) higher priority activities can block lower priority activities from being scheduled and (2) higher priority activities are more likely to appear in the schedule.

Mandatory Activities are activities, $m_1 \ldots m_j \subseteq A$, that must be scheduled. The presumption is that the problem as specified is *valid*, that is to say that a schedule exists that includes all of the mandatory activities, respects all of the provided constraints, and does not exceed available resources.

In addition, activities can be grouped into *Switch Groups*. The activities within a switch group are called *switch cases* and vary by how many resources (time, energy, and data volume) they consume. It is mandatory to schedule exactly one switch case and preferable to schedule a more resource intensive one, but not at the expense of another mandatory activity. For example, one of the Mars 2020 instruments takes images to fill mosaics which can vary in size; for instance we might consider $1x4$, $2x4$, or $4x4$ mosaics. Taking larger mosaics might be preferable, but taking a larger mosaic takes more time, takes more energy, and produces more data volume. These alternatives would be modeled by a switch group that might be as follows:

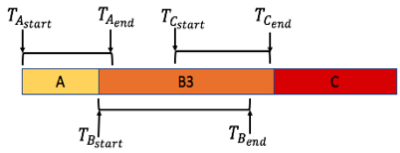
$$SwitchGroup = \begin{cases} Mosaic_{1x4} & d = 100 \text{ sec} \\ Mosaic_{2x4} & d = 200 \text{ sec} \\ Mosaic_{4x4} & d = 400 \text{ sec} \end{cases} \quad (1)$$

The desire is for the scheduler to schedule the activity $Mosaic_{4x4}$ but if it does not fit then try scheduling $Mosaic_{2x4}$, and eventually try $Mosaic_{1x4}$ if the other two fail to schedule. It is not worth scheduling a more consumptive switch case if doing so will prevent a future, lower priority mandatory activity from being scheduled due to lack of resources. Because our computationally limited scheduler cannot search or backtrack, it is a challenge to predict if a higher level switch case will be able to fit in the schedule without consuming resources that will cause another lower priority mandatory activity to be forced out of the schedule.

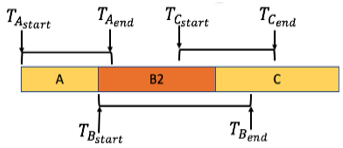
Consider the following example in Figure 1 where the switch group consists of activities B1, B2, and B3 and $d_{B3} > d_{B2} > d_{B1}$. Each activity in this example also has one start time window from $T_{i_{start}}$ to $T_{i_{end}}$.

B3 is the most resource intensive and has the highest priority so the scheduler will first try scheduling B3. As shown in Figure 1a, scheduling B3 will prevent the scheduler from placing activity C at a time satisfying its execution constraints. So, B3 should not be scheduled.

The question might arise as to why switch groups cannot simply be scheduled last in terms of scheduling order. This is difficult for several reasons: 1) We would like to avoid gaps in the schedule which is most effectively done by scheduling primarily left to right temporally, and 2) if another activity is dependent on an activity in a switch group, then scheduling the switch group last would introduce complications to ensure that the dependencies are satisfied.



(a) Scheduling B3 first prevents activity C from being scheduled within its start time window.



(b) B2 can be successfully scheduled without dropping any other mandatory activities.

Figure 1: Challenge to Schedule Switch Cases.

The remainder of the paper is organized as follows. First, we describe several plan wide energy constraints that must be satisfied. Then, we discuss two guard approaches to schedule preferred activities, which place conditions on the scheduler that restrict the placement of switch cases under certain conditions. We then discuss various versions of an approach which emulates backtracking by reinvoking the scheduler multiple times with the switch cases. We present empirical results to evaluate and compare these approaches.

Energy Constraints

There are several energy constraints which must be satisfied throughout scheduling and execution. The scheduling process for each *sol*, or Mars day, begins with the assumption that the rover is asleep for the entire time spanning the sol. Each time the scheduler places an activity, the rover must be awake so the energy level declines. When the rover is asleep the energy level increases.

Two crucial energy values which must be taken into account are the *Minimum State of Charge (SOC)* and the *Minimum Handover State of Charge*. The state of charge, or energy value, cannot dip below the Minimum SOC at any point. If scheduling an activity would cause the energy value to dip below the Minimum SOC, then that activity will not be scheduled. In addition, the state of charge cannot be below the *Minimum Handover SOC* at the *Handover Time*, in effect when the next schedule starts (e.g., the handover SOC of the previous plan is the expected beginning SOC for the subsequent schedule).

In order to preserve battery life, the scheduler must also consider the *Maximum State of Charge* constraint. Exceeding the Maximum SOC hurts long term battery performance and the rover will perform *shunting*. To prevent it from exceeding this value, the rover may be kept awake.

Guard Approaches

First we will discuss two guard methods to schedule switch cases, the Fixed Point guard and the Sol Wide guard. Both of these methods attempt to schedule switch cases by reserving enough time and energy to schedule the remaining mandatory activities. For switch groups, this means that resources will be reserved for the least resource consuming activity since it is mandatory to schedule exactly one activity in the switch group. The method through which both of these guard approaches reserve enough time to schedule future mandatory activities is the same. They differ in how they ensure there is enough energy. While the Fixed Point guard reserves enough energy at a single fixed time point - the time at which the least resource consuming switch case is scheduled to end in the nominal schedule, the Sol Wide guard attempts to reserve sufficient energy by keeping track of the energy balance in the entire plan, or sol.

In this discussion, we do not attempt to reserve data volume while computing the guards as it is not expected to be as constraining of a resource as time or energy. We aim to take data volume into account as we continue to do work on this topic.

Both the time and energy guards are calculated offline before execution occurs using a nominal schedule. Then, while rescheduling during execution, the constraints given by the guards are applied to ensure that scheduling a higher level switch case will not prevent a future mandatory activity from being scheduled. If activities have ended sufficiently early and freed up resources, then it may be possible to reschedule with a more consumptive switch case.

Guarding for Time

First, we will discuss how the Fixed Point and Sol Wide guards ensure enough time will be reserved to schedule remaining mandatory activities while attempting to schedule a more resource consuming switch case.

If a preferred time, $T_{i_{j\_preferred}}$, is specified for an activity, the scheduler will try to place an activity closest to its preferred time while obeying all other constraints. Otherwise, the scheduler will try to place the activity as early as possible.

Each switch group in the set of activities used to create a *nominal schedule* includes only the nominal, or least resource consuming switch case, and all activities take their predicted duration. First, we generate a nominal schedule and find the time at which the nominal switch case is scheduled to complete, as shown in Figure 2.

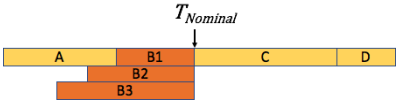


Figure 2: A, B1, C, and D are all mandatory activities in the nominal schedule. $T_{Nominal}$ is the time at which B1 is scheduled to end.

We then manipulate the execution time constraints of the more resource intensive switch cases, B2 and B3 in Figure 2, so that they are constrained to complete by $T_{Nominal}$ as shown in Equation 2. Thus, a more (time) resource consuming switch case will not use up time from any remaining lower priority mandatory activities. If an activity has more than one start time window, then we only alter the one which contains $T_{Nominal}$ and remove the others. If a prior activity ends earlier than expected during execution and frees up some time, then it may be possible to schedule a more consumptive switch case while obeying the time guard given by the altered execution time constraints.

$$T_{B_{ij\_end}} = T_{Nominal} - d_{B_i} \tag{2}$$

Since we found that the above method was quite conservative and heavily constrained the placement of a more resource consuming switch case, we attempted a *preferred time method* to loosen the time guard. In this approach, we set the preferred time of the nominal switch case to its latest start time before generating the nominal schedule. Then, while the nominal schedule is being generated, the scheduler will try to place the nominal switch case as late as possible since the scheduler will try to place an activity as close to its preferred time as possible. As a result, $T_{Nominal}$ will likely be later than what it would be if the preferred time were not set in this way. As per Equation 2, the latest start times, $T_{B_{ij\_end}}$, of the more resource consuming switch cases may be later than what they would be using the previous method where the preferred time was not altered, thus allowing for wider start time windows for higher level switch cases. This method has some risks. If the nominal switch case was placed as late as possible, it could use up time from another mandatory activity with a tight execution window that it would not otherwise have used up if it was placed earlier, as shown in Figure 3.

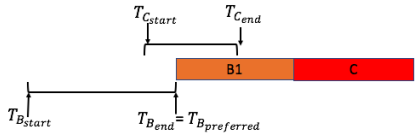


Figure 3: Scheduling B1 at its latest start time prevents C from being scheduled within its start time window.

Guarding for Energy

Fixed Point Minimum State of Charge Guard The Fixed Point method attempts to ensure that scheduling a more resource consuming switch case will not cause the energy to violate the Minimum SOC while scheduling any future mandatory activities by reserving sufficient energy at a single, fixed point in time, $T_{Nominal}$ as shown in Figure 4. The guard value for the Minimum SOC is the state of charge value at $T_{Nominal}$ while constructing the nominal schedule. When attempting to schedule a more resource intensive switch case, a constraint is placed on the scheduler so that the energy cannot fall below the Minimum SOC guard value at time $T_{Nominal}$. If an activity ends early (and uses

fewer resources than expected) during execution, it may be possible to satisfy this guard while scheduling a more consumptive switch case.

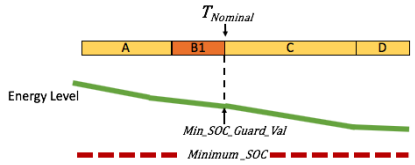


Figure 4: A, B1, C, and D, are mandatory activities in the nominal schedule. A constraint is placed so that the energy cannot dip below $Min\_SOC\_Guard\_Val$ at time $T_{Nominal}$ while trying to schedule a higher level switch case.

Fixed Point Handover State of Charge Guard The Fixed Point method guards for the Minimum Handover SOC by first calculating how much extra energy is left over in the nominal schedule at handover time after scheduling all activities, as shown in Figure 5.

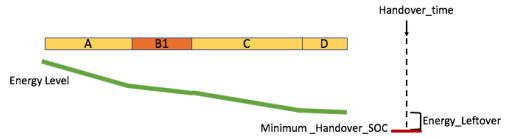


Figure 5: A, B1, C, and D, are mandatory activities in the nominal schedule. A constraint is placed so that the extra energy a higher level switch case consumes cannot exceed *Energy_Leftover*.

Then, while attempting to place a more consumptive switch case, a constraint is placed on the scheduler so that the extra energy required by the switch case does not exceed *Energy_Leftover* from the nominal schedule as in Figure 5. For example, if we have a switch group consisting of three activities, B1, B2, and B3 and $d_{B3} > d_{B2} > d_{B1}$ and each switch case consumes e Watts of power, we must ensure that the following inequality holds at the time the scheduler is attempting to schedule a higher level switch case:

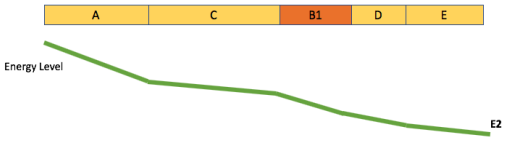
$$(d_{B_i} \times e_{B_i}) - (d_{B_1} \times e_{B_1}) \geq Energy\_Leftover \quad (3)$$

There may be more than one switch group in the schedule. Each time a higher level switch case is scheduled, the *Energy_Leftover* value is decreased by the extra energy required to schedule it. When the scheduler tries to place a switch case in another switch group, it will check against the updated *Energy_Leftover*.

Sol Wide Handover State of Charge Guard The Sol Wide handover SOC guard only schedules a more resource consumptive switch case if doing so will not cause the energy to dip below the Handover SOC at handover time. First, we use the nominal schedule to calculate how much energy is needed to schedule remaining mandatory activities. Having a Maximum SOC constraint while calculating this value may produce an inaccurate result since any energy that would exceed the Maximum SOC would not be taken into account. So, in order to have an accurate prediction of the energy balance as activities are being scheduled, this value is calculated assuming there is no Maximum SOC constraint. 8. The Maximum SOC constraint is only removed while computing the guard offline to gain a clear understanding of the energy balance but during execution it is enforced

As shown in Figure 6, the energy needed to schedule the remaining mandatory activities is the difference between the energy level just after the nominal switch case has been scheduled, call this E1, and after all activities have been scheduled, call this energy level E2.

(a) E1 is the energy level of the nominal schedule with no Maximum SOC constraint after all activities up to and including the nominal switch case (A, D, B1) have been scheduled.



(b) E2 is the energy level of the nominal schedule with no Maximum SOC constraint after all activities in the nominal schedule have been scheduled. The activities were scheduled the following order: A, D, B1, C, E.

Figure 6: Calculating Energy Needed to Schedule Remaining Mandatory Activities.

$$Energy\_Needed = E1 - E2 \quad (4)$$

Then, a constraint is placed on the scheduler so that the energy value after a higher level switch case is scheduled must be at least:

$$Energy\_Level \geq Minimum\_Handover\_SOC + Energy\_Needed \quad (5)$$

By placing this energy constraint, we hope to prevent the energy level from falling under the Minimum Handover SOC by the time all activities have been scheduled.

Sol Wide Minimum State of Charge Guard While we ensure that the energy will not violate the minimum Handover SOC by keeping track of the energy balance, it is possible that scheduling a longer switch case will cause the energy to fall below the Minimum SOC. To limit the chance of this happening, we run a Monte Carlo of execution offline while computing the sol wide energy guard. We use this Monte Carlo to determine if a mandatory activity was

not scheduled due to a longer switch case being scheduled earlier. If this occurs in any of the Monte Carlos of execution, then we increase the guard constraint in Equation 5. We first find the times at which each mandatory activity was scheduled to finish in the nominal schedule. Then, we run a Monte Carlo of execution with the input plan containing the guard and all switch cases. Each Monte Carlo differs in how long each activity takes to execute compared to its original predicted duration in the schedule. If a mandatory activity was not executed in any of the Monte Carlo runs and a more resource consuming switch case was executed before the time at which that mandatory activity was scheduled to complete in the nominal schedule, then we increase the Sol Wide energy guard value in Equation 5 by a fixed amount. We aim to compose a better heuristic to increase the guard value as we continue work on this subject.

Multiple Scheduler Invocation Approach

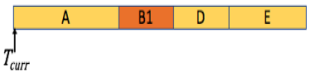
The Multiple Scheduler Invocation (MSI) approach emulates backtracking by reinvoking the scheduler multiple times with the switch cases. MSI does not require any precomputation offline before execution as with the guards and instead reinvokes the scheduler multiple times during execution. During execution, the scheduler reschedules (e.g., when activities end early) with only the nominal switch case as shown in Figure 7a until an MSI trigger is satisfied. At this point, the scheduler is reinvoked multiple times, at most once per switch case in each switch group. In the first MSI invocation, the scheduler attempts to schedule the highest level switch case as shown in Figure 7b. If the resulting schedule does not contain all mandatory activities, then the scheduler will attempt to schedule the next highest level switch case, as in 7c, and so on. If none of the higher level switch cases can be successfully scheduled then the schedule is regenerated with the nominal switch case. If activities have ended early by the time MSI is triggered and resulted in more resources than expected, then the goal is for this approach to generate a schedule with a more consumptive switch case if it will fit (assuming nominal activity durations for any activities that have not yet executed).

There are multiple factors that must be taken into consideration when implementing MSI:

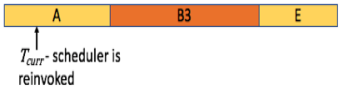
When to Trigger MSI There are two options to trigger the MSI process (first invocation while trying to schedule the switch case):

1. *Time Offset.* Start MSI when the current time during execution is some fixed amount of time, X, from the time at which the nominal switch case is scheduled to start in the current schedule (shown in Figure 8).
2. *Switch Ready.* Start MSI when an activity has finished executing and the nominal switch case activity is the next activity scheduled to start (shown in Figure 9).

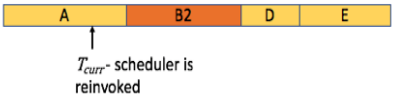
Spacing Between MSI Invocations If the highest level switch case activity is not able to be scheduled in the first invocation of MSI, then the scheduler must be invoked again. We choose to reschedule as soon as possible after the most recent MSI invocation. This method risks over-consumption



(a) MSI has not yet begun. Currently, the nominal switch case, B1, is scheduled.



(b) MSI begins. Scheduling the highest level switch case, B3, prevents D from being scheduled. Therefore, try B2.



(c) B2 is successfully scheduled along with the other mandatory activities so MSI is complete.

Figure 7: Order of MSI Invocations.

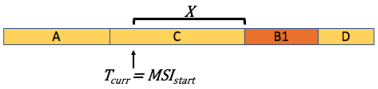


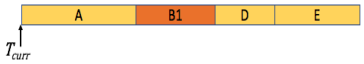
Figure 8: MSI Time Offset.

of the CPU if the scheduler is invoked too frequently. To handle this, we may need to rely on a process within the scheduler called *throttling*. Throttling places a constraint which imposes a minimum time delay between invocations, preventing the scheduler from being invoked at too high of a rate. An alternative is to reschedule at an evenly split, fixed cadence to avoid over-consumption of the CPU; we plan to explore this approach in the future.

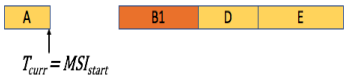
Switch Case Becomes Committed In some situations, the nominal switch case activity in the original plan may become committed before or during the MSI invocations as shown in Figure 10. An activity is *committed* if its scheduled start time is between the start and end of the commit window (Chien et al. 2000). A committed activity cannot be rescheduled and is committed to execute. If the nominal switch case remains committed, the scheduler will not be able to elevate to a higher level switch case.

There are two ways to handle this situation:

1. *Commit the activity.* Keep the nominal switch case activity committed and do not try to elevate to a higher level switch case.
2. *Veto the switch case.* Veto the nominal switch case so that it is no longer considered in the current schedule. When an activity is vetoed, it is removed from the current schedule and will be considered in a future invocation of the scheduler. Therefore, by vetoing the nominal switch case,



(a) B1 is the nominal switch case. Since an activity has not finished executing and B1 is not the next activity, MSI cannot begin yet.



(b) Since A finished executing early, and B1 is the next activity, the MSI process can begin.

Figure 9: MSI Switch Ready.

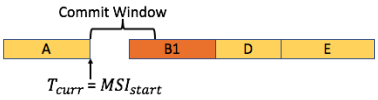


Figure 10: Switch case is committed during MSI. T_{curr} is the current time during execution. MSI_{start} is the time at which MSI begins. The nominal switch case, B1, is committed when MSI begins.

it will no longer be committed and the scheduler will continue the MSI invocations in an effort to elevate the switch case.

Handling Rescheduling After MSI Completes but before the Switch Case is Committed After MSI completes, there may be events that warrant rescheduling (e.g., an activity ending early) before the switch case is committed. When the scheduler is reinvoked to account for the event, it must know which level switch case to consider. If we successfully elevated a switch case, we choose to reschedule with that higher level switch case. Since the original schedule generated by MSI with the elevated switch case was in the past and did not undergo changes from this rescheduling, it is possible the schedule will be inconsistent and may lead to complications while scheduling later mandatory activities. An alternative we plan to explore in the future is to disable rescheduling until the switch case is committed. However, this approach would not allow the scheduler to regain time if an activity ended early and caused rescheduling.

Empirical Analysis

In order to evaluate the performance of the above methods, we apply them to various sets of inputs comprised of activities with their constraints and compare them against each other. The inputs are derived from *sol types*. *Sol types* are currently the best available data on expected Mars 2020 rover operations (Jet Propulsion Laboratory 2017a). In order to construct a schedule and simulate plan execution, we use the *Mars 2020 surrogate scheduler* - an implementation of the same algorithm as the Mars 2020 onboard scheduler (Rabideau and Benowitz 2017), but intended for a Linux workstation environment. As such, it is expected to produce the same schedules as the operational scheduler but runs much faster in a workstation environment. The surrogate scheduler is expected to assist in validating the flight scheduler implementation and also in ground operations for the mission (Chi et al. 2018).

Each sol type contains between 20 and 40 activities. Data from the Mars Science Laboratory Mission (Jet Propulsion Laboratory 2017b; Gaines et al. 2016a; 2016b) indicates that activity durations were quite conservative and completed early by around 30%. However, there is a desire by the mission to operate with a less conservative margin to increase productivity. In our model to determine activity execution durations, we choose from a normal distribution where the mean is 90% of the predicted, nominal activity duration. The standard deviation is set so that 10 % of activity execution durations will be greater than the nominal duration. For our analysis, if an activity's execution duration chosen from the distribution is longer than its nominal duration, then the execution duration is set to be the nominal duration to avoid many complications which result from activities running long (e.g., an activity may not be scheduled solely because another activity ran late). Detailed discussion of this is the subject of another paper. We do not explicitly change other activity resources such as energy and data volume since they are generally modeled as rates and changing activity durations implicitly changes energy and data volume as well.

We create 10 variants derived from each of 8 sol types by adding one switch group to each set of inputs for a total of 80 variants. The switch group contains three switch cases, $A_{nominal}$, A_{2x}, and A_{4x} where $d_{A_{4x}} = 4 \times d_{A_{nominal}}$ and $d_{A_{2x}} = 2 \times d_{A_{nominal}}$.

In order to evaluate the effectiveness of each method, we have developed a scoring method based on how many and what type of activities are able to be scheduled successfully. The score is such that the value of any single mandatory activity being scheduled is much greater than that of any combination of switch cases (at most one activity from each switch group can be scheduled).

Each mandatory activity that is successfully scheduled, including whichever switch case activity is scheduled, contributes one point to the *mandatory score*. A successfully scheduled switch case that is 2 times as long as the original activity contributes $1/2$ to the *switch group score*. A successfully scheduled switch case that is 4 times as long as the original, nominal switch case contributes 1 to the switch group score. If only the nominal switch case is able to be scheduled, it does not contribute to the switch group score at all. There is only one switch group in each variant, so the maximum switch group score for a variant is 1. Since scheduling a mandatory activity is of much higher importance than scheduling any number of higher level switch case, the mandatory activity score is weighted at a much larger value then the switch group score. In the following empirical results, we average the mandatory and switch groups scores over 20 Monte Carlo runs of execution for each variant.

We compare the different methods to schedule switch cases over varying incoming state of charge values (how much energy exists at the start) and determine which methods result in 1) scheduling all mandatory activities and 2) the highest switch group scores. The upper bound for the theoretical maximum switch group score is given by an *omniscient scheduler*- a scheduler which has prior knowledge of the execution duration for each activity. Thus, this scheduler is aware of the amount of resources that will be available to schedule higher level switch cases given how long activities take to execute compared to their predicted, nominal duration. The input activity durations fed to this omniscient scheduler are the actual execution durations. We run the omniscient scheduler at most once per switch case. First, we try to schedule with only the highest level switch case and if that fails to schedule all mandatory activities, then we try with the next level switch case, and so on.

First, we determine which methods are able to successfully schedule all mandatory activities, indicated by the Maximum Mandatory Score in Figure 11. Since scheduling a mandatory activity is worth much more than scheduling any number of higher level switch cases, we only compare switch group scores between methods that successfully schedule all mandatory activities.

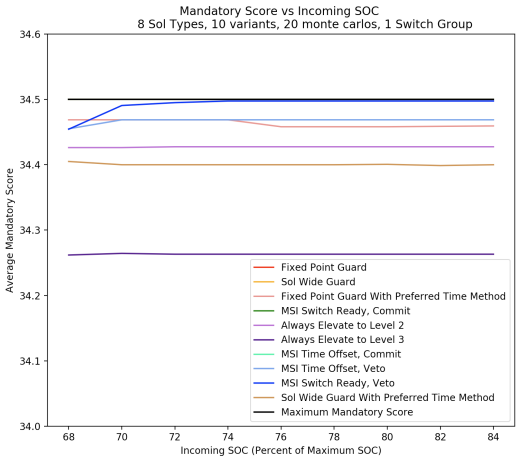


Figure 11: Mandatory score vs Incoming SOC for various Methods to Schedule Switch Cases

In order to evaluate the ability of each method to schedule all mandatory activities, we also compare against two other methods, one which always elevates to the highest level switch case while the other always elevates to the medium level switch case. We see in Figure 11 that always elevating to the highest (3rd) level performs the worst and drops approximately 0.25 mandatory activities per sol, or 1 activity per 4 sols on average while always elevating to the second highest level drops close to 0.07 mandatory activities per sol, or 1 activity per 14 sols on average. For comparison, the study described in (Gaines et al. 2016a) showed that approximately 1 mandatory activity was dropped every 90 sols, indicating that both of these heuristics perform poorly.

We found that using preferred time to guard against time

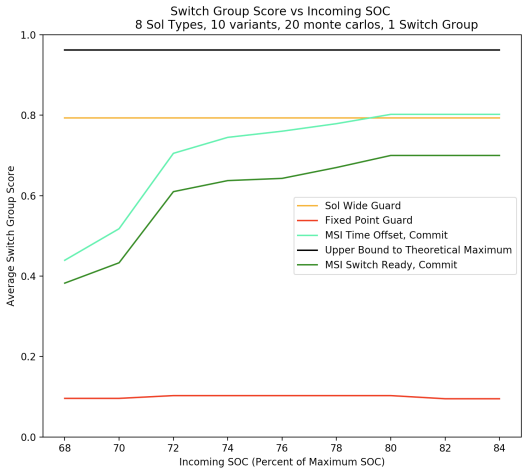


Figure 12: Switch Group Score vs Incoming SOC for Methods which Schedule all Mandatory Activities

caused mandatory activities to drop for both the fixed point and sol wide guard (for the reason described in the Guarding for Time section) while using the original method to guard against time did not. We see in Figure 11 that the preferred time method with the fixed point guard drops on average about 0.04 mandatory activities per sol, or 1 activity every 25 sols while the sol wide guard drops on average about 0.1 mandatory activities per sol, or 1 activity every 10 sols. We also see that occasionally fewer mandatory activities are scheduled with a higher incoming SOC. Since using preferred time does not properly ensure that all remaining activities will be able to be scheduled, a higher incoming SOC can allow a higher level switch case to be scheduled, preventing future mandatory activities from being scheduled.

The MSI approaches which veto to handle the situation where the nominal switch case becomes committed before or during MSI drop mandatory activities. Whenever an activity is vetoed, there is always the risk that it will not be able to be scheduled in a future invocation, more so if the sol type is very tightly time constrained, which is especially true for one of our sol types. Thus, vetoing the nominal switch case can result in dropping the activity, accounting for this method's inability to schedule all mandatory activities. The MSI methods that keep the nominal switch case committed and do not try to elevate to a higher level switch case successfully schedule all mandatory activities, as do the guard methods.

We see that the Fixed Point guard, Sol Wide guard, and two of the MSI approaches are able to successfully schedule all mandatory activities. As shown in Figure 12, the Sol Wide guard and MSI approach using the options Time Offset and Commit result in the highest switch group scores closest to the upper bound for the theoretical maximum. Both MSI approaches have increasing switch group scores with increasing incoming SOC since a higher incoming energy will result in more energy to schedule a consumptive switch case during MSI. The less time there is to complete all MSI

invocations, the more likely it is for the nominal switch case to become committed. Since we give up trying to elevate switch cases and keep the switch case committed if this occurs, fewer switch cases will be elevated. Because our time offset value, X, in Figure 8 is quite large (15 minutes), this situation is more likely to occur using the Switch Ready approach to choose when to start MSI, explaining why using Switch Ready results in a lower switch score than Time Offset.

The Fixed Point guard results in a significantly lower switch case score because it checks against a state of charge constraint at a particular time regardless of what occurs during execution. Even if a switch case is being attempted to be scheduled at a completely different time than $T_{Nominal}$ in Figure 2, (e.g., because prior activities ended early), the guard constraint will still be enforced at that particular time. Since we simulate activities ending early, more activities will likely complete by $T_{Nominal}$, causing the energy level to fall under the Minimum SOC Guard value. Unlike the Fixed Point guard, since the the Sol Wide guard checks if there is sufficient energy to schedule a higher level switch case at the time the scheduler is attempting to schedule it, not at a set time, it is better able to consider resources regained from an activity ending early.

We also see that using the Fixed Point guard begins to result in a lower switch group score with higher incoming SOC levels after the incoming SOC is 80% of the Maximum SOC. Energy is more likely to reach the Maximum SOC constraint with a higher incoming SOC. The energy gained by an activity taking less time than predicted will not be able to be used if the resulting energy level would exceed the Maximum SOC. If this occurs, then since the extra energy cannot be used, the energy level may dip below the guard value in Figure 4 at time $T_{Nominal}$ while trying to schedule a higher level switch case even if an activity ended sufficiently early, as shown in Figure 13.

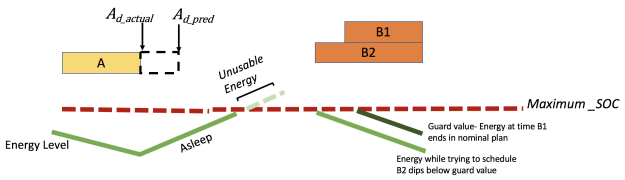


Figure 13: Fixed Point Guard Schedules Fewer Mandatory Activities with Higher Incoming SOC

Related Work

Just-In-Case Scheduling (Drummond, Bresina, and Swanson 1994) uses a nominal schedule to determine areas where breaks in the schedule are most likely to occur and produces a branching (tree) schedule to cover execution contingencies. Our approaches all (re) schedule on the fly although the guard methods can be vewied as forcing schedule branches based on time and resource availability.

Kellenbrink and Helber (Kellenbrink and Helber 2015) solve RCPSP (resource-constrained project scheduling problem) where all activities that must be scheduled are not known in advance and the scheduler must decide whether or not to perform certain activities of varying resource consumption. Similarly, our scheduler does not know which of the switch cases to schedule in advance, using runtime resource information to drive (re) scheduling.

Integrated planning and scheduling can also be considered scheduling disjuncts (chosen based on prevailing conditions (e.g., (Barták 2000))) but these methods typically search whereas we are too computationally limited to search.

Discussion and Future Work

There are many areas for future work. Currently the time guard heavily limits the placement of activities. As we saw, using preferred time to address this issue resulted in dropping mandatory activities. Ideally analysis of start time windows and dependencies could determine where an activity could be placed without blocking other mandatory activities.

Additionally, in computing the guard for Minimum SOC using the Sol Wide Guard, instead of increasing the guard value by a predetermined fixed amount which could result in over-conservatism, binary search via Monte Carlo analysis could more precisely determine the guard amount.

Currently we consider only a single switch group per plan, the Mars 2020 rover mission desires support for multiple switch groups in the input instead. Additional work is needed to extend to multiple switch groups.

Further exploration of all of the MSI variants is needed. Study of starting MSI invocations if an activity ends early by at least some amount and the switch case is the next activity is planned. We would like to analyze the effects of evenly spacing the MSI invocations in order to avoid relying on throttling and we would like to try disabling rescheduling after MSI is complete until the switch case has been committed and understand if this results in major drawbacks.

We have studied the effects of time and energy on switch cases, and we would like to extend these approaches and analysis to data volume.

Conclusion

We have presented several algorithms to allow a very computationally limited, non-backtracking scheduler to consider a schedule containing required, or mandatory, activities and sets of activities called switch groups where each activity in such sets differs only by its resource consumption. These algorithms strive to schedule the most preferred, which happens to be the most consumptive, activity possible in the set without dropping any other mandatory activity. First, we discuss two guard methods which use different approaches to reserve enough resources to schedule remaining mandatory activities. We then discuss a third algorithm, MSI, which emulates backtracking by reinvoking the scheduler at most once per level of switch case. We present empirical analysis using input sets of activities derived from data on expected planetary rover operations to show the effects of using each of these methods. These implementations and empirical evaluation are currently being evaluated in the context of the Mars 2020 onboard scheduler.

Acknowledgments

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

Barták, R. 2000. Conceptual models for combined planning and scheduling. *Electronic Notes in Discrete Mathematics* 4(1).

Chi, W.; Chien, S.; Agrawal, J.; Rabideau, G.; Benowitz, E.; Gaines, D.; Fosse, E.; Kuhn, S.; and Biehl, J. 2018. Embedding a scheduler in execution for a planetary rover. In *ICAPS*.

Chien, S. A.; Knight, R.; Stechert, A.; Sherwood, R.; and Rabideau, G. 2000. Using iterative repair to improve the responsiveness of planning and scheduling. In *Artificial Intelligence Planning and Schedling*, 300–307.

Drummond, M.; Bresina, J.; and Swanson, K. 1994. Just-in-case scheduling. In *AAAI*, volume 94, 1098–1104.

Gaines, D.; Anderson, R.; Doran, G.; Huffman, W.; Justice, H.; Mackey, R.; Rabideau, G.; Vasavada, A.; Verma, V.; Estlin, T.; et al. 2016a. Productivity challenges for mars rover operations. In *Proceedings of 4th Workshop on Planning and Robotics (PlanRob)*, 115–125. London, UK.

Gaines, D.; Doran, G.; Justice, H.; Rabideau, G.; Schaffer, S.; Verma, V.; Wagstaff, K.; Vasavada, A.; Huffman, W.; Anderson, R.; et al. 2016b. Productivity challenges for mars rover operations: A case study of mars science laboratory operations. Technical report, Technical Report D-97908, Jet Propulsion Laboratory.

Jet Propulsion Laboratory. 2017a. Mars 2020 rover mission https://mars.nasa.gov/mars2020/ retrieved 2017-11-13.

Jet Propulsion Laboratory. 2017b. Mars science laboratory mission https://mars.nasa.gov/msl/ 2017-11-13.

Kellenbrink, C., and Helber, S. 2015. Scheduling resource-constrained projects with a flexible project structure. *European Journal of Operational Research* 246(2):379–391.

Rabideau, G., and Benowitz, E. 2017. Prototyping an onboard scheduler for the mars 2020 rover. In *International Workshop on Planning and Scheduling for Space*.

On Expected Value Strong Controllability

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

Abstract

The Probabilistic Simple Temporal Network (PSTN) generalizes Simple Temporal Networks with Uncertainty (STNUs) by introducing probability distributions over the timing of uncontrollable timepoints. PSTNs are controllable if there is a strategy to execute the controllable timepoints while minimizing or bounding the risk of violating any constraint. If the risk is too high, PSTNs are not considered controllable. We introduce the Expected Value Probabilistic Simple Temporal Network (EPSTN), which extends PSTNs by including a benefit to the satisfaction of temporal constraints. We study the problem of Expected Value Strong Controllability (EvSC) of EPSTNs, which seeks a schedule maximizing the expected value of satisfied constraints. We solve the EvSC problem using a mixed integer linear program (MILP), combined with search over constraints to violate at execution time. The MILP bounds below the probability of satisfying constraints; we quantify the error of the lower bound. We then show how to search for constraints to discard, using the MILP at the core of the search. While the general problem is shown to be exponential, we conclude by providing methods to bound the complexity of search.

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 executed while satisfying temporal constraints in the presence of uncertain outcomes? Many previously studied solutions to this problem use the notion of controllability of Simple Temporal Networks under Uncertainty (STNUs) at their core. The solutions to such problems are *strategies* to execute all events that ensure no constraints are violated, regardless of the outcomes of uncertain events. More complex problems combine temporal constraints, uncertainty, and *preferences*. Tractability of these problems is ensured due to the simplicity of the constraints and preferences provided as inputs. Uncertainty can be generalized so that algorithms must handle *probabilities* over when events occur; these networks are referred to as Probabilistic Simple Temporal Networks (PSTNs). This leads to new *risk-bounded* and *chance-constrained* problems. If the solution is still unsatisfactory, these constraints can be *relaxed* using a cost functions on the constraints and risk bound.

What happens when it is *almost certain* that a constraint will be violated? A review of the recent PSTN literature shows that many instances in current benchmarks are either not controllable, or are controllable but with high risk. For instance, consider the CAR-SHARING benchmark of (Fang, Yu, and Williams 2014): only 184 of 1800 instances are strongly controllable, about 10%. For the ROVERS benchmark of (Santana et al. 2016): optimistically, 2840 of 4380 instances are strongly controllable, but in 911 of these cases, at least one probabilistic duration squeezed to *a single value*. A more accurate assessment is that 1929 of of the instances are strongly controllable, or about 44%. Finally, if we consider the PSTNs analyzed in (Lund et al. 2017), less than 20% of the 540 instances are strongly controllable.

Current approaches, particularly risk-bounding, do not adequately address the problem of uncontrollable PSTNs. If the risk bound is too low, no strategy can be produced. If a risky strategy is produced, then an undesirable outcome at execution time will violate a constraint, causing unexpected execution-time (e.g. 'freezing' execution, damaging the system, etc.). When the existing set of constraints cannot be satisfied to produce control strategies, relaxing some constraints up-front may ensure controllability, but with a loss of insight into the original problem. An alternative solution to such over-constrained problems is to let the execution strategy try to satisfy as many constraints as possible, assuming that at least one constraint will be violated during execution. If some constraints are more important than others, then a natural optimization criterion for the strategy is to maximize the *expected value* of satisfied constraints. This new, unexplored 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.

In this paper we define the Expected Value Probabilistic Simple Temporal Network EPSTN (EPSTN). We formalize the problem of finding a fixed schedule maximizing the expected value of satisfied constraints, the Expected Value Strong Controllability (EvSC) problem. We adapt algorithms from the controllability literature to solve this problem, and provide soundness and completeness results. Solving the general problem is shown to require exponen-

tial time; we conclude by providing methods to bound the complexity of search.

2 Notation and Definitions

In this section we will define controllability problems previously considered in the literature. These definitions will set the stage for the formalization of our new class of problem.

Definition 1 (STNU)**.** *(Vidal and Ghallab 1996) (Vidal and Fargier 1999) (Muscettola, Morris, and Vidal 2001) Simple Temporal Networks with Uncertainty (STNUs) consist of* Controllable *time-points, $A = \cup_i a_i$, i.e. those assigned by the agent, and* Uncontrollable *time-points, $R = \cup_i r_i$, i.e. those assigned by the external world. The set of timepoints $T = A \cup R$. The domain of $t_i \in T = \mathbb{R}$. Denote by $v(r_i)$ the observed value of r_i during execution.* Requirement 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* $\exists\, v(r_i) \in [l_{a_i,r_j}, u_{a_i,r_j}] \mid r_j - a_i = v(r_i)$, *but $v(r_i)$ 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 2 (Strong Controllability)**.** *(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 outcomes of all contingent constraints). A schedule s is an assignment to $a_i \in A$. Denote the value of a_i in s by $s(a_i)$. P is* Strongly Controllable *(SC) if there is a schedule s such that $\forall v \in V$, s satisfies all constraints $c(t_i, t_j)$.*

Definition 3 (PSTN)**.** *(Tsamardinos 2002) Let a* probabilistic duration constraint $d(a_i, r_j)$ *have the form* $r_j - a_i = \omega_i \in \Omega_{a_i,r_j}$ *where $a_i \in A$, $r_j \in R$, and Ω_{a_i,r_j} is a random variable with probability distribution function $P(\Omega_{a_i,r_j})$. Let $D = \cup_{a_i,r_j} d(a_i, r_j)$. A Probabilistic Simple Temporal Networks (PSTN) is a 4-tuple <A,R,C,D>.*

In the sequel, we will assume w.l.o.g. that there is a 1-1 mapping between probabilistic duration constraints and controllable timepoints, i.e. $\forall\{d(a_i, r_k), d(a_j, r_m)\}$, $a_i \neq a_j$, allowing us to say $r_j - a_i \in \Omega_j$, and also allowing us to write the bounds as $[l_{r_j}, u_{r_j}]$ (but see Section 4.4).

Risk as introduced by (Fang, Yu, and Williams 2014) describes the probability that, given a schedule or strategy, an outcome $v \in V$ violates some constraint. Typical approaches transform a PSTN into an STNU and then evaluate controllability. To compute risk for an STNU, we measure how much probability mass on each uncertain duration is not covered after 'squeezing' to transform it into a contingent link, i.e. transforming $d(a_i, r_j)$ to $g(a_i, r_j)$, in a manner similar to (Santana et al. 2016). The definitions below bound above the risk, because our definition of PSTNs does not assume that the probabilities are independent.

Definition 4. *Let ρ_d:D⇒G transform a duration constraint into a contingent link by choosing a compact subset* $[l_{r_j}, u_{r_j}] \subset \Omega_j$. *Let $\rho_D = \{\rho_d\}$. Let P be a PSTN. Then $\rho_D(P) = U$ where U is the STNU derived from P.*

Definition 5. *Let P be a PSTN. Let $U = \rho_D(P)$ be an STNU derived from P. Let $\rho_d(d(a_i, r_i)) = g(a_i, r_i)$. Let $[l_{r_i}, u_{r_i}]$ be the contingent constraint interval defined by $g(a_i, r_i)$. Let $\Phi_g = \omega \in \Omega_i | \omega \leq l_{r_i}$. Let $\Theta_g = \omega \in \Omega_i | \omega \geq u_{r_i}$. The* risk *of $d(a_i, r_i)$ relative to ρ_d, denoted $\delta(\rho_d, d(a_i, r_i))$, is $\int_{\omega \in \Phi_g \cup \Theta_g} P(\Omega_i)$. The* risk *of P relative to ρ_D, denoted $\delta(P, \rho_D)$, is bounded above by $1 - \left(\prod_{d \in D}(1 - \delta(\rho_d, d(t_i, t_j)))\right)$.*

Definition 6. *P is SC with risk $\leq \Delta$ if $\exists\, P' = \rho_D(P)$, P' is SC, and $\delta(P, \rho_D) = \Delta$.*

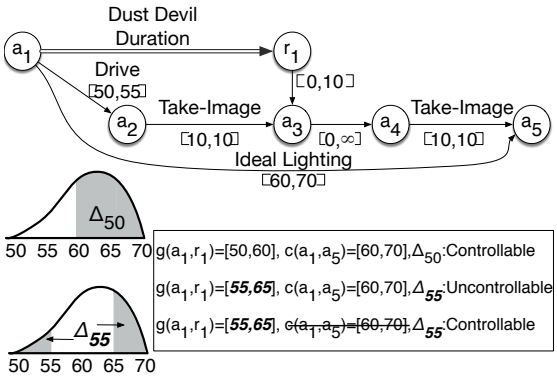


Figure 1: Sample Problem showing how sacrificing a constraint can reduce risk.

3 A Running Example and Previous Work

Consider a planetary exploration rover with two imaging goals: one is a dynamic phenomenon of uncertain duration (a dust devil), and the second is an image of a static target, but with a constraint driven by ideal lighting conditions. The images take 10 minutes to collect. The dust devil is expected to last $50 - 70$ minutes after the drive starts, and the drive to a position from which the dust devil can be imaged can be planned for between 50 and 55 minutes. The ideal lighting for the second image occurs between 60 to 70 minutes after starting the drive. The PSTN is shown in Figure 1. As in Definition 1, denote the observed value of r_1 by $v(r_1)$. If $g(a_1, r_1) = [50, 60]$ then the resulting STNU is strongly controllable; the schedule $s(a_2)$=$s(a_1)$+50, $s(a_3)$=$s(a_1)$+60, $s(a_4)$=$s(a_1)$+60, $s(a_5)$=$s(a_1)$+70 is valid for any value of $r_1 \in [50, 60]$. In order to transform this PSTN into an STNU, we search over $\rho(d(a_1, r_1)) = g(a_1, r_1)$ to satisfy an initially aggressive risk bound, Δ_{50}. Suppose we deem the resulting risk of $v(r_1) > s(a_3)$, which violates $c(r_1, a_3)$, to be too high; we prefer a lower risk option, e.g. $s(a_2) = 55$ satisfying Δ_{55}, allowing $g(a_1, r_1) = [55, 65]$. Unfortunately, any assignment $s(a_2) > 50$ ultimately violates $c(a_1, a_5)$; with the lower risk bound, the resulting STNU is not strongly controllable.

One previously explored approach for such problems is to search over *relaxations* for a problem that can be transformed into a controllable STNU with some bounded risk; (Yu, Fang, and Williams 2015) use this approach for conditional STNUs. The search is guided by costs of relaxations of either the requirement constraints or the risk bound. In the example above, a *relaxed* constraint $c(a_1, a_5) = [60, 75]$ (not

shown) would lead to a controllable STNU with $s(a_2)= 55$. While this approach bounds the likelihood of violating the relaxed constraints in the transformed STNU at execution time, the original constraints are lost, so there is no information to guide generation of the strategy to avoid violating constraints unnecessarily. Modeling allowable constraint violations could be addressed by first relaxing the bounds on the requirement constraints, and adding *preferences* that value satisfying the original constraint more than the relaxed bounds. This could be done using simple semi-convex preference functions, combined with 'min', to achieve tractability, as in (Rossi, Venable, and Yorke-Smith 2006). However, this approach is too limiting; in particular, the 'min' function will report the worst preference achieved for any constraint, which could be 0 (representing a 'violated' constraint).

In our example, we would like to evaluate trading satisfaction of the lighting constraint $c(a_1, a_5)$ with the dust devil observation constraint, $c(r_1, a_3)$. The right strategy depends on the relative importance of satisfying $c(r_1, a_3)$ and $c(a_1, a_5)$, and the probability of satisfying $c(r_1, a_3)$, which requires formulating the *expected value* of a schedule or strategy. The expected value formulation is common in MDPs; while the relaxation approach in (Yu, Fang, and Williams 2015) minimizes the cost of relaxations, it does not maximize the expected value of the controllability strategy. The continuous time nature of the state space precludes using formulations such as time-dependent MDPs (Boyan and Littman 2000); the desire to express state spaces representing violated constraints makes other time-based MDP approaches e.g. (Weld and Mausam 2006) inappropriate.

4 The EPSTN

We now formalize the Expected Value Probabilistic Simple Temporal Network (EPSTN) by adding constraint valuations $q_c(t_i, t_j)$, to a PSTN. We then formalize the Expected Value Strong Controllability (EvSC) problem on EPSTNs.

Definition 7 (EPSTN)**.** *Let $q_c(t_i, t_j)$: $c(t_i, t_j) \Rightarrow \mathbb{R}^+$ and let Q be the set of all $q_c(t_i, t_j)$. An Expected Value Probabilistic Simple Temporal Network (EPSTN) is a 5-tuple <A,R,C,D,Q>.*

Definition 8. *Let P_e be an EPSTN. Let s be a schedule. Let $\sigma_c(t_i, t_j, s, v) \Rightarrow \{0, 1\}$ be 1 if $c(t_i, t_j)$ is satisfied by (v, s) (by extracting $v(r_i)$ for $t_i = r_i$ or $s(a_j)$ for $t_i = a_i$ and evaluating the bounds) and 0 otherwise. Then $f_{P_e}(s, v) = \sum_{c \in C} q_c(t_i, t_j)\ (\sigma_c(t_i, t_j, s, v))$ is the value of a schedule s combined with a set of outcomes $v \in V$. The expected value of s is then $E(f_{P_e}(s, V)) = \int_{v \in V}(P(v) f_{P_e}(s, v))$. Given an EPSTN, the* Expected Value Strong Controllability *(EvSC) problem is to find s maximizing $E(f_{P_e}(s, V))$.*

While a similar Dynamic Controllability problem can also be formalized, for the remainder of the paper, we will focus on Expected Value Strong Controllability.

EPSTNs are a variant of the Disjunctive Temporal Problem with Preferences (DTPP) (Peinter, Moffitt, and Pollack 2005), in which not all constraints can be satisfied, and the 'best' set must be found by search. Each requirement constraint can be expressed as a disjunction where satisfying the 'trivial' constraint has zero value and satisfying the original constraint has value $q_c(t_i, t_j)$. The EPSTN is a *strict generalization* of the DTPP; while the value of satisfying $c(a_i, a_j)$ is captured by $q_c(a_i, a_j)$, the expected value of satisfying $c(r_i, a_j)$ is a nontrivial function of timepoint assignments, rather than a constant associated with the disjunctive decisions. EPSTNs are also similar to the Controllable Conditional Temporal Problem with Uncertainty (CCTPU) of (Yu, Fang, and Williams 2015), in that we can choose which constraints to satisfy. EPSTNs are more general than CCTPUs in that they include preferences, but are more limited in that every timepoint of an EPSTN must be scheduled.

We now look deeper at the fundamental tradeoff in EvSC: sacrificing a constraint to improve the overall expected value of a schedule. In Figure 1 above, there is only one requirement constraint over an uncontrollable timepoint, namely $c(r_1, a_3)$. If $50 \leq v(r_1) \leq 55$ we can construct a schedule violating a single constraint, namely, $c(a_1, a_5)$, in order to satisfy $c(r_1, a_3)$ and ensure all other constraints are satisfied. Committing to a schedule up-front that violates $c(a_1, a_5)$ lets us increase the probability $c(r_1, a_3)$ is satisfied, potentially increasing the expected value of the schedule. We can determine the *relative* values of $q_c(r_1, a_3)$ and $q_c(a_1, a_5)$ that make violating $c(a_1, a_5)$ maximize the expected value. Assume $s(a_1)= 0$. Let s be a schedule in which $s(a_2)= 50$ and s' be a schedule in which $s(a_2)= 55$. For s' to be preferred, we would need

$$q_c(r_1, a_3) \int_0^{50} P(\Omega_1) + q_c(a_1, a_5) < q_c(r_1, a_3) \int_0^{55} P(\Omega_1)$$
$$\Rightarrow q_c(a_1, a_5) < q_c(r_1, a_3) \left(\int_0^{55} P(\Omega_1) - \int_0^{50} P(\Omega_1) \right)$$
$$\Rightarrow q_c(a_1, a_5) < q_c(r_1, a_3) \int_{50}^{55} P(\Omega_1)$$

If $\int_{50}^{55} P(\Omega_1)$ is 'small', the inequality above is only satisfied if $q_c(r_1, a_3)$ is 'large'. Put another way, $q_c(r_1, a_3)$ needs to be a factor of $\frac{1}{\int_{50}^{55} P(\Omega_1)}$ larger than $q_c(a_1, a_5)$. Committing to bounds on contingent links that maximize the expected value for a high-value constraint on an uncontrollable may violate other constraints. We may intuitively view this as creating a *cycle* that must be broken by deleting a low-value constraint. It may be necessary to remove multiple constraints to increased coverage of a single high-value uncontrollable duration. Breaking each cycle may reveal another cycle involving another requirement constraint, as shown in Figure 2 (left). In this example, it is obvious that breaking the minimum cost edge unexpectedly *decreases* the expected value. Even if removing a series of constraints decreases the risk, the expected value can still decrease, until removing enough constraints leads to an eventual *net increase* in the expected value. This scenario is shown in Figure 2 (right). The situation becomes more complex when we consider that removing a single constraint might lead to decreased risk, and therefore expected value, of multiple high-value constraints.

Our roadmap for EvSC is as follows. We first describe how to bound below the expected value of an EPSTN while enforcing *all* requirement constraints over two controllable timepoints. These are referred to as the Simple Temporal Network (STN) constraints, since they are constraints found in STNs. We solve the more general EPSTN problem by searching over subsets of the STN constraints to enforce;

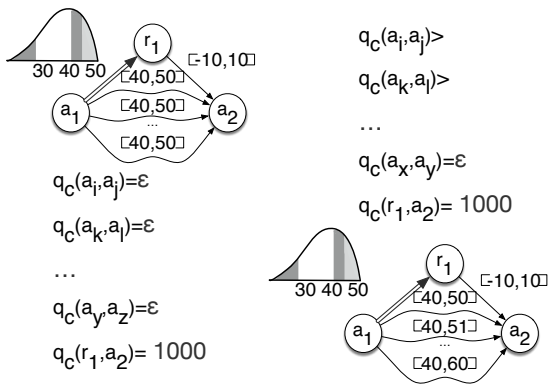


Figure 2: Multiple constraints may need to be removed to ultimately increase expected value, leading to a series of reductions in expected value before it improves after all cycles are broken (left). Removing a constraint and decreasing the risk may still reduce the expected value (right).

as we saw above, the value of the STN constraints must be traded against the expected value of satisfying At-Risk (AR) requirement constraints on an uncontrollable timepoint. Along the way, we provide some insights into algorithm complexity, soundness and completeness of these approaches, which shed light on the difficulty of addressing expected controllability.

4.1 Semi-Simple EPSTNs

We begin with some definitions:

Definition 9 (Simple and Semi-Simple EPSTN)**.** *Let P_e be an EPSTN. Denote the STN constraints $c(a_i, a_j)$ by C_s. Denote the At-Risk (AR) constraints $c(r_i, a_j)$ by C_u. P_e is* Semi-Simple *if $\exists$ s that satisfies all constraints in C_s. P_e is* Simple *if it is Semi-Simple and if, $\forall P'_e$ defined by C_s' $\subset C_s$, $\max_s E(f_{P_e}(s, V)) \geq \max_s E(f_{P'_e}(s, V))$.*

The EPSTN in Figure 1 is Semi-Simple, because the STN constraints alone (all requirement constraints except $c(r_1, a_3)$) are satisfiable. If $\frac{1}{\int_{50}^{55} P(\Omega_1)} q_c(r_1, a_3) < q_c(a_1, a_5)$, then the original EPSTN is also Simple, because removing any set of STN constraints does not increase the expected value; otherwise, removing $c(a_1, a_5)$ makes the resulting EPSTN Simple.

Suppose we fix, or are otherwise given, a set of STN constraints over controllable timepoints C_s that can all be satisfied. We start with an EPSTN P_e and generally don't know the set C_s' that will lead to optimality, nor do we know the schedule of maximum value. Thus it appears two simultaneous searches are required: a subset C_s' leading to optimality, and the best SC schedule (that is, the one maximizing the expected value of satisfied constraints) for P'.

SREA (Lund et al. 2017) uses a series of linear programs (LPs) that are constructed and solved in order to minimize the risk of a PSTN; a feasible LP is guaranteed to be SC. However, we can't use the SREA LP formulation. First, it does not explicitly represent the risk. Second, we don't merely want to minimize the risk that the outcomes don't respect the bounds of the contingent constraints, but rather maximize the expected value of a schedule. Specifically, we need to explicitly represent the probability that an uncontrollable event's actual time $v(r_i) = s(a_i) + \omega_i \in \Omega_i$ leads to a violation of AR constraint $c(r_i, a_j)$. This means we can't directly use $\delta(\rho_d, d(a_i, r_i))$ from Definition 5, because this definition computes the risk by measuring how much of the probability mass of $d(a_i, r_i)$ is covered by $\rho_d(d(a_i, r_i))$.

When computing the expected values, it is helpful to imagine a triangle of two requirement constraints $c(a_i, a_j)$, $c(r_i, a_j)$, and a duration constraint $d(a_i, r_i)$. Since we have assumed P_e is Semi-Simple, we know there is a schedule s satisfying $c(a_i, a_j)$. Ideally, $v(r_i)$ and $s(a_j)$ will satisfy the constraint $c(r_i, a_j)$. But an outcome $v(r_i)$ may be unlucky, either violating the lower bound l_{r_i,a_j}, because a_i and a_j are scheduled close together to satisfy $c(a_i, a_j)$ and ω_i is too large, or violating the upper bound u_{r_i,a_j}, because a_i and a_j are scheduled far apart and ω_i is too small.

We exploit the fact that for a specific assignment $s(a_i), s(a_j)$, the probability of satisfying the constraint $c(r_i, a_j)$, and obtaining value $q_c(r_i, a_j)$, is
$\int_{s(a_j)-s(a_i)-u_{r_i,a_j}}^{s(a_j)-s(a_i)-l_{r_i,a_j}} P(\Omega_i)$

This may appear backwards, but it isn't. Consider the lower bound of the integral, $s(a_j) - s(a_i) - u_{r_i,a_j}$. The smallest value of ω_i satisfying $c(r_i, a_j)$ is $s(a_j) - s(a_i) - u_{r_i,a_j}$. The later $v(r_i) = s(a_i) + \omega_i$ occurs, the smaller $s(a_j) - s(a_i) - \omega_i$ is, therefore it is the lower bound of the integral. Similarly, the largest value of ω_i satisfying $c(r_i, a_j)$ is $s(a_j) - s(a_i) - l_{r_i,a_j}$. The earlier $v(r_i) = s(a_i) + \omega_i$ occurs, the larger $s(a_j) - s(a_i) - \omega_i$ is, therefore it is the upper bound of the integral.

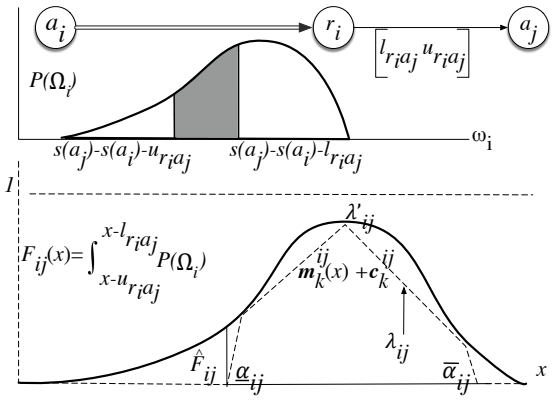


Figure 3: Bounding the probability to construct an MILP.

In general, we must search over many possible schedules, meaning $s(a_j) - s(a_i)$ will vary, and so will the value of the integral defining the probability of success. We want a declarative representation of the possible values of this integral to use during search; in particular, we would prefer not to perform integrals in the inner loop of this search. For a probabilistic duration $d(a_i, r_i)$ and linked requirement constraint $c(r_i, a_j)$ the function $F_{ij}(x) = \int_{x-u_{r_i,a_j}}^{x-l_{r_i,a_j}} P(\Omega_i)$ has as its only 'free' variable the distance $x = s(a_j) - s(a_i)$; the bounds from $c(r_i, a_j)$ are, effectively, constant parameters. If $P(\Omega_i)$ is unimodal, then $F_{ij}(x)$ is unimodal and rep-

resents all possible probability masses we might encounter during search. We would like solve the maximum expected value problem using an MILP; given the nonlinearity of $F_{ij}(x)$, we must in practice use a linear approximation. We choose to approximate from below to get a conservative estimate on the probability of constraint satisfaction. While $F_{ij}(x)$ is not concave over its entire range, we can bound its concave region (which will include its mode) below with a series of linear inequalities defined by functions $m_k^{ij}x + c_k^{ij}$ that collectively underestimate the probability of obtaining value $q_c(r_i, a_j)$. The free variable λ_{ij} in the MILP represents the probability of satisfying $c(r_i, a_j)$ given a specific distance $s(a_j) - s(a_i)$, and is bound above by this piecewise linear approximation. The construction is shown in Figure 3. We must construct at most $|C_u|$ functions F_{ij}, one per AR constraint. It appears we must perform many integrals to construct each F_{ij}. Once we have decided how many points each piecewise linear approximation will have, however, we can limit the number of integrals. We also have the benefit of being able to do definite integrals, which is often easy for 'nice' families (e.g. normal distributions.)

As shown in Figure 3, the piecewise linear bound will cross the x axis at points where $F_{ij}(x) > 0$. We enforce $\lambda_{ij} = 0$ when x falls on the left-hand or right-hand side of the "good" (i.e. concave) region using a single binary variable b_{ij}. If we let $\underline{\alpha}_{ij}$ and $\overline{\alpha}_{ij}$ represent the lower and upper bounds, respectively, on the "good" region, we can choose a sufficiently large constant $M > 0$ and add the constraints $\underline{\alpha}_{ij} - M(1 - b_{ij}) \leq s(a_j) - s(a_i) \leq \overline{\alpha}_{ij} + M(1 - b_{ij})$, bound λ_{ij} above by b_{ij}, and augment the right-hand sides of all constraints (6) by $M(1-b_{ij})$. The values for $\underline{\alpha}_{ij}$ and $\overline{\alpha}_{ij}$ that yield the widest "good" region are the x-intercepts of the lines tangent to F_{ij} at the left- and right-hand bounds of the region in which it is concave (i.e. its inflection points), but these may prove difficult to compute analytically and are thus approximated instead.

We construct the functions $m_k^{ij}x + c_k^{ij}$ and compute the values for $\underline{\alpha}_{ij}$ and $\overline{\alpha}_{ij}$ as follows. We first evaluate $F_{ij}(x)$ at a given number of equally spaced x-values in its range (the more evaluations, the better the approximation) and draw line segments between consecutive points. Then, starting from the left and moving right, we choose the left endpoint of the first line segment whose slope is not larger than that of its predecessor, and, starting from the right and moving left, we choose the right endpoint of the first line segment whose slope is not smaller (i.e. steeper) than that of its predecessor. These points will serve as our approximate left and right inflection points, and we use the x-intercepts of the lines tangent to F_{ij} at these points as our $\underline{\alpha}_{ij}$ and $\overline{\alpha}_{ij}$ values. Furthermore, the equations of the lines defining the segments found between these points together with those two tangent lines will serve as our set of bounding functions $m_k^{ij}x + c_k^{ij}$. With a fine enough discretization, these functions collectively define a good lower approximation of the concave part of F_{ij}.

The MILP is shown in Figure 4. The constants (indicated in bold in the MILP) are the $q_c(r_i, a_j)$, the requirement constraint bounds $[l_{a_i,a_j}, u_{a_i,a_j}]$, M, $\underline{\alpha}_{ij}$ and $\overline{\alpha}_{ij}$, and the slopes and intercepts of the piecewise linear approximation

$$\max \sum_{c(r_i,a_j)} \lambda_{ij} \mathbf{q_c}(\mathbf{r_i}, \mathbf{a_j})$$

$$\text{s.t. } a_0 = 0 \tag{1}$$

$$a_j - a_i \leq \mathbf{u_{a_i,a_j}} \qquad \forall c(a_i, a_j) \in C_s \tag{2}$$

$$a_j - a_i \geq \mathbf{l_{a_i,a_j}} \qquad \forall c(a_i, a_j) \in C_s \tag{3}$$

$$a_j - a_i \leq \overline{\boldsymbol{\alpha}}_{ij} + \mathbf{M}(1 - b_{ij}) \qquad \forall c(r_i, a_j) \in C_u \tag{4}$$

$$a_j - a_i \geq \underline{\boldsymbol{\alpha}}_{ij} - \mathbf{M}(1 - b_{ij}) \qquad \forall c(r_i, a_j) \in C_u \tag{5}$$

$$\lambda_{ij} \leq \mathbf{m_k^{ij}}(a_j - a_i) + \mathbf{c_k^{ij}} + \mathbf{M}(1 - b_{ij}) \qquad \forall c(r_i, a_j) \in C_u, \forall k = 1, \ldots, y_{ij} \tag{6}$$

$$\lambda_{ij} \leq b_{ij} \qquad \forall c(r_i, a_j) \in C_u \tag{7}$$

$$b_{ij} \in \{0, 1\} \qquad \forall c(r_i, a_j) \in C_u \tag{8}$$

Figure 4: MILP to maximize the expected value of a Semi-Simple EPSTN.

of $F_{ij}(x)$. The variables are the a_i representing the schedule for the controllables, λ_{ij} representing the probability of satisfying constraint $c(r_i, a_j)$, and b_{ij} representing whether the probability is forced to zero by dint of being to the left or right of the concave approximation of $F_{ij}(x)$. The objective function is now simply $\max \sum_{c(r_i,a_j)} \lambda_{ij} q_c(r_i, a_j)$.

4.2 Quality of the MILP Solution

As described in the previous section, the piecewise linear functions in constraints (6) and the binary variables b_{ij} together serve to bound the true probability of the schedule satisfying a given contingent constraint from below. Thus, the optimal solution to the MILP will generally lead to error in the expected value. To see that nontrivial error in solutions can occur, we observe that if the piecewise linear approximation is flat, leading to the same probability λ_{ij} in many MILP solutions, the true curve $F_{ij}(x)$ may yield better solutions only found by nonlinear optimization starting at the arbitrary solution found using the piecewise linear approximation. Note this will generally occur below $\underline{\alpha}_{ij}$ or above $\overline{\alpha}_{ij}$, but it could also occur near the mode of $F_{ij}(x)$ if it is approximated by only a small number of pieces. Let $\hat{G}_{ij} = \max_x F_{ij}(x) - (m_k^{ij}x + c_k^{ij})$ be the largest difference between the true value of $F_{ij}(x)$ and its piecewise linear lower bound. The worst-case error of any solution to any Semi-Simple EPSTN is $\hat{E} = \sum_{c(r_i,a_j)} q_c(r_i, a_j)\hat{G}_{ij}$.

Given a schedule s, let λ_{ij}^s be the probabilities of success for the schedule. The error $\hat{E}(s)$ of this schedule is $\sum_{c(r_i,a_j)} q_c(r_i, a_j) \left(F_{ij}\left(s(a_j) - s(a_i)\right) - \lambda_{ij}^s\right)$. Suppose that the true optimal schedule o with value O is found when we only enforce the subset C_s^o of STN constraints from P_e. Let $\hat{O} = O - \hat{E}(o)$, that is, the quality of the schedule according to the MILP. Let $q_c^o = \sum_{c(a_i,a_j) \in C_s^o} q_c(r_i, a_j)$. Suppose b is the optimal solution for the MILP over C_s^o, and $\hat{B}$ is the quality of b according to the MILP. We have $O \geq \hat{B} \geq \hat{O}$. Then

$$\begin{aligned} O &= q_c^o + \textstyle\sum_{c(r_i,a_j)} q_c(r_i, a_j) F_{ij}\left(o(a_j) - o(a_i)\right) \\ &= q_c^o + \textstyle\sum_{c(r_i,a_j)} q_c(r_i, a_j)\lambda_{ij}^o + \hat{E}(o) \end{aligned}$$

$\leq q_c^o + \sum_{c(r_i,a_j)} q_c(r_i, a_j)\lambda_{ij}^b + \hat{E}(o)$
$= \hat{B} + \hat{E}(o) \leq \hat{B} + \hat{E}$.

If our algorithm finds the overall best solution $(a, \hat{A})$ by removing some other subset $C_s' \neq C_s^o$ of STN constraints, then $\hat{B} \leq \hat{A}$, and therefore $O \leq \hat{A} + \hat{E}$. In general, computing this worst-case error $\hat{E}$ requires computing $\hat{G}_{ij}$ for all at-risk constraints $c(r_i, a_j)$. Given enough pieces, the approximation inside the concave region will be good enough that the error is maximized at the tails, and therefore we only need to compute the difference at $\underline{\alpha}_{ij}$ and $\overline{\alpha}_{ij}$ to find $\hat{G}_{ij}$.

4.3 Complexity, Soundness and Incompleteness

The resulting MILP has $|A|$ continuous variables a_i for controllable timepoint assignments, $|C_u|$ continuous variables λ_{ij} approximating the probabilities of satisfying at-risk constraints, and $|C_u|$ binary variables b_{ij} to enforce appropriate upper bounding of these λ_{ij} values, for totals of $|A| + |C_u|$ continuous and $|C_u|$ binary variables. It has $2|C_s|$ constraints enforcing the lower and upper bounds of requirement constraints, $2|C_u|$ constraints checking whether or not $a_j - a_i$ values fall within their "good" regions, $|C_u|$ constraints bounding λ values from above within their "bad" regions, and no more than $y_*|C_u|$ constraints bounding λ_{ij} values from above within their "good" regions, where $y_* = \max_{c(r_i,a_j)} y_{ij}$, for a total of no more than $2|C_s|+(3+y_*)|C_u|$ constraints. Solving MILPs is known to be $\mathcal{NP}$-complete, but it is difficult to formally characterize the complexity of solving this MILP as it depends not only on the number of binary variables in the formulation but also on the tightness of the lower bound provided by solving its LP relaxation (which in our case won't be great since we're using "big-M" constraints). Nevertheless, we have the following result:

Lemma 1. *Given an EPSTN P_e, a schedule b satisfying all constraints in C_s whose expected value $\hat{B}$ is within $\hat{E}$ of the true optimal expected value over C_s can be found in $O(2^{|C_u|}|T|^3)$.*

Given a Semi-Simple EPSTN P_e, there must exist a Simple EPSTN P_e^o with $C_s^o \subseteq C_s$ (even if $C_s^o = \emptyset$). Clearly, the optimal solution can be found by removing *every* subset of the set of requirement constraints C_s in turn and using the MILP described above as a sub-solver after each removal. If we enumerate each of the exponentially many subsets of STN constraints and solve the MILP for the resulting EPSTN as described above, we can simply perform an exhaustive search, always keeping track of the best solution and corresponding expected value found so far and updating when necessary. The objective function in the MILP omits the value of the STN constraints since any feasible solution will satisfy them. Thus to properly compare solutions with different sets of active STN constraints we must simply add the values of all active STN constraints to the value we get from solving the MILP. This is consistent, in general, with results for DTPPs (Peinter, Moffitt, and Pollack 2005). In conjunction with the error bound, the discussion above proves the following:

Theorem 1. *Given an EPSTN P_e, with (unknown) maximum expected value O. Our algorithm returns a solution $(a, \hat{A})$ in $O(|T|^3 2^{|C|})$ time such that $O \leq \hat{A} + \hat{E}$.*

Thus, the exponential search described above is *sound* in that it will return a feasible schedule bounding below the maximum expected value, and *complete* insofar as it will return the *best* such solution given our piecewise linear approximation of the distributions of uncontrollable durations of the EPSTN.

4.4 Applicability of the Algorithm

We have not explicitly assumed that any pair of probabilities $P(\Omega_i)$ and $P(\Omega_j)$ in an EPSTN are *conditionally independent*. If the uncertain durations are not independent, then the allocation of risk will generally increase the optimality gap. To see this, note that a schedule captures risk in an n-dimensional box, while correlated risk distributions could be covered by other 'shapes' with less restrictive constraints.

Handling the requirement constraints on two uncontrollables, $c(r_i, r_j)$, requires generalizing the $F_{ij}(x)$ to $\int_{x-u_{r_i,r_j}}^{x-l_{r_i,r_j}} P(\Omega_j) - P(\Omega_i)$. Even if $P(\Omega_i)$ and $P(\Omega_j)$ are unimodal, the composition in general is not (Ibragimov 1956), and thus free constraints $c(r_i, r_j)$ are not permitted in our EPSTNs [1]. The same proviso applies when reformulating linked duration constraints $d(a_i, r_i)$ and $d(r_i, r_j)$ [2]. We must replace $d(r_i, r_j)$ with $d(a_k, r_k)$ such that $P(\Omega_k) = P(\Omega_i) + P(\Omega_j)$ and $[l_{a_i,a_k}, u_{a_i,a_k}] = [0, 0]$ [3].

We also note that while our function bounding the probability of satisfaction from below does not define a concave region (and thus requires that a binary "on-off" variable be added to what was a nice LP), it gives us the power to consider situations in which all possible outcomes $v(r_i)$ for an uncontrollable must fall entirely on one side of its mode μ_i. This was one of the difficulties in dealing with unimodal distributions that the authors in (Santana et al. 2016) did not address, as their choice of functions used to approximate risk required that the bounds for acceptable outcomes for an uncontrollable fall on either side of its mode. Our increased coverage comes at a price: the PARIS algorithm in (Santana et al. 2016) is polynomial-time, while even our 'inner-loop' algorithm for semi-simple EPSTNs is exponential (we must solve $O(2^{|C_s|})$ MILPs.)

5 Testing for Simplicity

The search described above can be implemented as a tree search over sets of requirement constraints to exclude from our EPSTN in the quest for the schedule maximizing the

[1] The sum, and thus difference, of two independent normally distributed variables is a normal, and thus unimodal; under these and similar conditions the formulation provided works and $c(r_i, r_j)$ are permitted. Otherwise, constraints $c(r_i, r_j)$ and preferences $q_c(r_i, r_j)$ can be modeled, with some difficulty, by two constraints $c(a_k, r_i)$ and $c(a_k, r_j)$.

[2] Allowed by (Santana et al. 2016) but not (Tsamardinos 2002).

[3] If we have $d(a_i, r_k)$ and $d(a_i, r_m)$, we can replace these constraints with $d(a_i, r_k), d(a_j, r_m)$ and $c(a_i, a_j)$ with $[l_{a_i,a_j}, u_{a_i,a_j}] = [0, 0]$, which causes no difficulty.

expected value. An inexpensive test for simplicity can be used to terminate the exponential search described above, and potentially reduce search time. This test leverages the existence of an optimal solution for a Semi-Simple EPSTN.

Consider an optimal schedule b for a Semi-Simple EPSTN P_e, and consider the triangle formed by constraints $g(a_i, r_i)$, $c(r_i, a_j)$ and $c(a_i, a_j)$. From the construction of the MILP, we know the expected value of AR constraint $c(r_i, a_j)$ in b is $q_c(r_i, a_j)\ \lambda^b_{ij}$, where λ^b_{ij} is bounded above by $F_{ij}(x)$. Let $\lambda'_{ij} = \max_x F_{ij}(x)$. This is the maximum possible probability of obtaining $q_c(r_i, a_j)$ and can be obtained as we construct $F_{ij}(x)$. (A tight constraint $c(r_i, a_j)$ combined with a large variance on $P(\Omega_i)$ will lead to $\lambda'_{ij} < 1$.) The sacrifice of any STN constraint can only improve the expected value of this AR constraint by at most $q_c(r_i, a_j)$ $(\lambda'_{ij} - \lambda^b_{ij})$. Is it possible for the removed constraint to be 'accidentally' satisfied by a schedule s' maximizing the expected value for the relaxed problem, while the expected value on the AR constraints is increased? If so, then this hypothetical schedule s' would have larger value for the original EPSTN than the optimal schedule b, which is a contradiction. This leads to the following definition:

Definition 10 (Gain)**.** *Given a Semi-Simple EPSTN P_e. Let λ^b_{ij} be the value of λ_{ij} in the optimal solution b to the MILP for P_e. We define the* gain $\gamma(P_e, b) = \sum_{c(r_i,a_j)\in C_u} q_c(r_i, a_j)\ (\lambda'_{ij} - \lambda^b_{ij})$.

By construction, the gain for each possible optimal solution will be identical. Even so, the values of λ^b_{ij} can vary across solutions. As we will now see, it is convenient to define the gain in terms of the values of λ^b_{ij} in the solution b. A straightforward test for simplicity follows:

Theorem 2. *Given a semi-simple EPSTN P_e, and the optimal solution b to the MILP forP_e. Then P_e is Simple if $\gamma(P_e, b) \leq \min_{c(a_i,a_j)\in C_s} q_c(a_i, a_j)$.*

The above test determines whether sacrificing the *least* valuable STN constraint can (optimistically) improve the expected value by *all* of the gain achievable. If even this optimistic tradeoff is not favorable, then removing more STN constraints can't improve the expected value, therefore P_e is Simple. The test is necessary but not sufficient; the test may fail when P_e is simple. In particular, if we look at Figure 1, we see that removing $c(a_5, a_6)$ cannot relax $c(r_1, a_1)$ because the two constraints are not on the same cycle. However, if $q_c(a_5, a_6) \geq \gamma(P_e, s)$, the test would fail, and search might fruitlessly attempt to relax $c(a_5, a_6)$.

5.1 Bounding the Search Costs

It is tempting to think that exponential search for the best subset of C_s in a semi-simple EPSTN is unnecessary. Unfortunately, as we saw above in the test for simplicity, in general this will not be the case; one might need to search over edges shared between cycles. However, we can use the gain to evaluate the *largest set* of STN constraints whose sacrifice could, possibly, be offset by the gain. The size of this set can be used to bound the search cost.

Theorem 3. *Given a semi-simple EPSTN P_e, and the optimal solution b to the MILP for P_e. Let $C_{s,\gamma} \subset C_s$ be $c(a_i, a_j) \in C_s \mid q_c(a_i, a_j) \leq \gamma(P_e, b)$. Let $M \subset C_{s,\gamma}$ be the largest cardinality set such that $\sum_{c\in M} q_c(a_i, a_j) \leq \gamma(P_e, b)$. Then the search cost cannot exceed $O\left(|T|^3 2^{|C_u|} \sum_{k=1}^{|M|} \binom{|C_{s,\gamma}|}{k}\right)$.*

Proof. Definition 10 allows us to immediately reduce C_s to $C_{s,\gamma}$. To compute the largest cardinality set M: sort $q_c(a_i, a_j)$ in increasing order. Begin with an empty set. Add $c(a_i, a_j)$ to the set until the next largest $q_c(a_i, a_j)$ would produce a set M such that $\sum_{c_{a_i a_j}\in M} q_c(a_i, a_j) > \gamma(P_e, s)$. The gain is non-increasing as search proceeds: each time we throw an STN constraint away, λ_{ij} cannot decrease (but could increase). Further, STN constraints are only removed from P_e so they can't ever be added to M above. So M at the beginning of search is the biggest it will ever get.

The procedure only identifies the largest sized set; there can be other sets with different constraints (example: $\gamma(P_e, s)$ is 1000, there are 1000 constraints of quality 1 and one constraint of quality 500.) The number of sets whose size is $\leq |M|$ is $\sum_{k=1}^{|M|} \binom{|C_{s,\gamma}|}{k}$. Since removal of each set requires solving the MILP, search therefore takes $O(|T|^3 2^{|C_u|} \left(\sum_{k=1}^{|M|} \binom{|C_{s,\gamma}|}{k}\right))$. □

We simplify $\sum_{k=1}^{|M|} \binom{|C_{s,\gamma}|}{k})$ as follows: Assume $|M| \leq \frac{|C_{s,\gamma}|}{2}$. Then

$\sum_{k=1}^{|M|} \binom{|C_{s,\gamma}|}{k}) \leq |M|\binom{|C_{s,\gamma}|}{|M|}) \leq |M||C_{s,\gamma}|^{|M|}$

$= |M|(2^{|M|\log|C_{s,\gamma}|})$

The case of $|M| \geq \frac{|C_{s,\gamma}|}{2}$ requires bounding above $\sum_{k=1}^{|M|} \binom{|C_{s,\gamma}|}{k})$ and subtracting from $2^{|C_s|}$, but is similar. While the bound is loose, this analysis shows that considerable reduction in search cost can be achieved by using the gain. When $\gamma(P_e, s)$ is large relative to the average value of $q_c(a_i, a_j)$, then $|M|$ will be large; when $\gamma(P_e, s)$ is small, then $|M|$ will be small. Theorem 3 will generally overestimate the exponential costs of search. This is both because of the loose bound on the number of sets of constraints whose total value is bounded above by $\gamma(P_e, s)$, and because the bound does not take into account whether the requirement constraints' values that contribute to large $|M|$ will actually lead to increased utility by loosening constraints on an uncertain duration.

5.2 Search Algorithm

We now present the search algorithm to bound below the maximum expected value schedule given a Semi-Simple EPSTN P_e. The algorithm is given in Algorithm 1. The basic strategy is to perform tree search over the largest cardinality set M such that $\sum_{c\in M} q_c(a_i, a_j) \leq \gamma(P_e, s)$. This is done by choosing the minimum preference constraint to consider for elimination at each search step. Both induced EPSTNs are generated, one forced to include this constraint and one with this constraint deleted. To simplify the pseudocode, we augment the objective function of the MILP to represent the expected schedule value including the STN constraints; it is now $\sum_{c(a_i,a_j)\in C_s} q_c(a_i, a_j) + \sum_{c(r_i,a_j)\in C_u} (q_c(r_i, a_j)\lambda_{ij})$.

The recursion terminates when Theorem 2 applies and a Simple EPSTN is detected, i.e. search continues only when $q_{c_m} = \min_{c(a_i,a_j)\in C_s} q_c(a_i, a_j) \leq \gamma(P_e, s)$. A 'branch and bound' test ensures that continued search has a chance of improving on the best expected value found so far, i.e. search terminates when $(\hat{S} - q_{c_m} + \gamma(P_e, s) \geq \hat{B})$. This second test only passes when Theorem 2 applies.

The maximum expected value schedule over all of these induced EPSTNs is returned when we initialize the algorithm with $K = \emptyset$, $\hat{B} = 0$, and b an arbitrary schedule (feasible or infeasible). Preventing needless re-solving of the MILP on the second recursive call is a minor modification that can be made to improve efficiency.

Algorithm 1: MaxEPSTN

Input : An EPSTN, P_e
Input : Constraints considered for removal, K
Input : Current Best Schedule / Expected Value $(b, \hat{B})$
Output: Best Schedule / Expected Value pair $(b, \hat{B})$
Var : LP representation of P_e, LP_e
Var : Minimum preference constraint, c_m
Var : Temp schedule and value, $(s, \hat{S})$

$(LP_e) \leftarrow$ `MakeLP`(P_e) ;
$(s, \hat{S}) \leftarrow$ `Solve` (LP_e) ;
if $(\hat{S} > \hat{B})$ **then**
  $(b, \hat{B}) \leftarrow (s, \hat{S})$;
if $((C_s \setminus K) \neq \emptyset)$ **then**
  $c_m \leftarrow \arg\min_{c(a_i,a_j)\in(C_s\setminus K)} q_c(a_i, a_j)$;
  if $(\hat{S} - q_{c_m} + \gamma(P_e, s) \geq \hat{B}))$ **then**
    $K \leftarrow K \cup c_m$;
    $(s, \hat{S}) \leftarrow$ `MaxEPSTN` $(P_e - c_m, K, (b, \hat{B}))$;
    if $\hat{S} > \hat{B}$ **then**
      $(b, \hat{B}) \leftarrow (s, \hat{S})$;
    $(s, \hat{S}) \leftarrow$ `MaxEPSTN` $(P_e, K, (b, \hat{B}))$;
    if $\hat{S} > \hat{B}$ **then**
      $(b, \hat{B}) \leftarrow (s, \hat{S})$;
return $(b, \hat{B})$;

6 Conclusions and Future Work

When presented with a control problem on probabilistic simple temporal networks, the usual strategy of establishing controllability may fail when constraints are too stringent. To address this, we formally define a new type of controllability problem, the Expected Value Probabilistic Simple Temporal Network (EPSTN), and address the Expected Value Strong Controllability (EvSC) problem of finding a schedule maximizing the expected value of satisfied constraints. We first formulate an MILP to find a schedule for a special case in which all STN constraints must be satisfied. The expected value of this schedule bounds below the true expected value using a piecewise linear approximation of the probability of satisfying the AR constraints in the EPSTN. This MILP must be solved at each branch of a search that discards STN constraints to allow covering more and more probability mass. Search can be pruned using termination rules that guarantee no favorable cost-benefit tradeoffs remain to be explored. While this search is exponential in the number of simple temporal constraints in the EPSTN, we bound the exponent by reasoning about the tradeoff between the lost value of each constraint and the expected gain.

The next step in evaluating the EvSC algorithm presented in this paper is to perform an empirical study. The datasets described in the introduction will form the basis of such a study, but they need to be augmented by addition of $q_c(r_i, a_j)$. The EvSC problem formulation requires careful choice of preferred STN constraints $c(a_i, a_j)$, such as coordinated observations or 'ideal' constraints on event time.

Our EvSC algorithm can be improved by exploring heuristics for selecting STN constraints to eliminate; these heuristics may detect cycles in the constraint graph of the resulting STN to eliminate even more search candidates. Incremental solving of the MILP may also help reduce search costs. However, the EvSC problem can also be solved by using a single MILP to handle both STN constraints and AR constraints. Preliminary results show that this is a promising approach. A nice benefit of this approach is that it generalizes easily to handle cases in which we must also search to find a subset of satisfiable STN constraints, as in DTPPs (Peinter, Moffitt, and Pollack 2005).

The error bound $\hat{E}$ can be reduced once a specific solution a is found. To see this, realize that the true error $\hat{E}(a)$ can be found at the cost of a few extra integrations to determine the true probabilities of success, rather than the piecewise linear bound. Thus, solution quality is within $\hat{E} - \hat{E}(a)$ of the true optimal expected value O. Other error bound improvements possible.

The simple case of fixed-value preferences $q_c(t_i, t_j)$ can be extended to preferences over intervals, as is done for Simple Temporal Problems with Preferences (STPPs) (Rossi, Venable, and Yorke-Smith 2006). By making some reasonable assumptions on the shape of these preferences, much of the theory described in this paper can be reused, leading to similar algorithms and computational complexity results.

The Expected Value Dynamic Controllability (EvDC) problem on EPSTNs remains open. Solving this problem will provide executives with the ability to respond dynamically to unexpected outcomes in order to maximize the expected value of satisfied constraints, which existing risk-bounding and constraint-relaxing strategies simply cannot do. The solution to this problem is likely to be quite different than the techniques described in this work.

References

Boyan, J., and Littman, M. 2000. Exact solutions to time-dependent MDPs. In *NIPS*, 1026–1032.

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.

Ibragimov, I. A. 1956. On the composition of unimodal distributions. *Teor. Veroyatnost. i Primenen.* 1(2):283–288.

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.

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*.

Peinter, B.; Moffitt, M. D.; and Pollack, M. E. 2005. Solving overconstrained disjunctive temporal problems with preferences. In *Proceedings of the* 15^{th} *International Conference on Automated Planning and Scheduling*.

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.

Weld, D., and Mausam. 2006. Probabilistic temporal planning with uncertain durations. In *Proceedings of the National Conference on Artificial Intelligence*, 880 – 887.

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

Dynamic Controllability with Single and Multiple Indirect Observations

Paul Morris
NASA Ames Research Center
Moffett Field, CA 94035, U.S.A.

Arthur Bit-Monnot
University of Sassari
Sassari, Italy

Abstract

A recent paper introduced a transformation-based approach for determining dynamic controllability of Simple Temporal Networks with Uncertainty (STNUs) extended to have variably-delayed observations of uncontrolled timepoints. Although the approach correctly determines dynamic controllability, it does not always provide the most flexible possible dynamic strategy. We show how to refine the approach in a way that improves the flexibility, and further extend it to a class of Partially Observable STNUs where the hidden timepoints can be indirectly observed via a chain of contingent links. We show how to construct a labeled distance graph for these problems, leading to a complete solution. This approach handles "single-headed" chained contingent links. For "multi-headed" problems, we prove a theorem characterizing their dynamic controllability in isolation. This provides a check on more general networks (and more general methods). We also consider potential extensions of the single-headed approach to multi-headed problems and point out some difficulties that arise.

Introduction

The Simple Temporal Network (STN) formalism introduced by Dechter, Meiri, and Pearl (Dechter, Meiri, and Pearl 1991) has proved very fruitful for reasoning about temporal plans. It has been extended in various directions including the STNU formalism that deals with external events whose timing is uncertain (Vidal and Fargier 1999; Morris, Muscettola, and Vidal 2001; Hunsberger 2009; Morris 2014), and effective algorithms have been developed to handle these problems. An additional extension (Moffitt 2007) introduced the Partially Observable STNU (POSTNU) formalism that may include uncontrolled timepoints that can be observed only indirectly, through their subsequent effects on other timepoints that are observable. In this paper, the uncontrolled timepoints that cannot be directly observed may conveniently be called *hidden* timepoints.

A recent paper (Bhargava, Muise, and Williams 2018) introduces a related extension of STNU, called Variable Delay STNU, where an uncontrollable event is determined to have occurred only after some delay whose duration is itself uncertain. It was noted that a Variable Delay STNU can be modeled as a special case of a POSTNU where the observational delay is represented as a separate contingent link that is activated by the uncontrollable event that is being observed. Thus, viewed as a POSTNU, the network involves two chained contingent links. The uncontrolled timepoint that is observed only indirectly (via the chained link) is a hidden timepoint. The Variable Delay paper considers networks that may have additional requirement constraints on the hidden timepoints. It introduces novel methods to tranform such Variable Delay STNUs into an STNU where contingent timepoints are either instantaneously observable or never observable. The result is essentially a POSTNU without chained constraints, which allows dynamic controllability to be determined by existing methods (Bit-Monnot, Ghallab, and Ingrand 2016). It also presents an execution strategy for networks that are found to be dynamically controllable.

The method presented in the Variable Delay paper correctly determines dynamic controllability and presents a valid execution strategy. However, the strategy presented is not always the most flexible possible, as we will show by an example. This can be remedied by a "doubling" strategy where the timepoint following a hidden timepoint is split in two. We will show how the doubling arises naturally in a first principles analysis. Also, for a Variable Delay STNU viewed as a POSTNU, a hidden timepoint activates at most one chained constraint. This may be described as a "single-headed" chained constraint. We will consider "multi-headed" cases where several contingent links are activated by the same hidden timepoint, and prove a theorem relating the dynamic controllability of a multi-headed network fragment to a relationship between the "slack" (upper bound minus lower-bound) of the requirement and contingent constraints involved. This result can be used as a check on the validity of transformation methods. Our results extend the set of cases where algorithms for DC checking of POSTNU are complete.

STNUs and Extensions

A Simple Temporal Network (STN) (Dechter, Meiri, and Pearl 1991) is a graph in which the edges are annotated with upper and lower numerical bounds. The nodes in the graph represent temporal events or *timepoints*, while the edges (refered to as *links*) correspond to constraints on the durations between the events. For instance a link A $\xrightarrow{[2,5]}$ B imposes that at least 2 time units and no more than 5 time

units elapse between the occurrence of A and the occurence of B. Each STN is associated with a *distance graph* where each link A $\xrightarrow{[x,y]}$ B is replaced by two edges A $\xrightarrow{y}$ B and A $\xleftarrow{-x}$ B. An STN is consistent if and only if the distance graph does not contain a negative cycle.

A Simple Temporal Network With Uncertainty (STNU) is similar to an STN except the links are divided into two classes, *requirement links* and *contingent links*. Requirement links are temporal constraints that the agent must satisfy, like the links in an ordinary STN. Contingent links may be thought of as representing causal processes of uncertain duration, or periods from a reference time to exogenous events; their finish timepoints, called here *contingent timepoints*, are controlled by Nature, subject to the limits imposed by the bounds on the contingent links. All other timepoints, called *executable timepoints*, are controlled by the agent, whose goal is to satisfy the bounds on the requirement links. The start timepoint of a contingent link is called its *activation timepoint* and can be either contingent or executable. Each contingent link is required to have non-negative (finite) upper and lower bounds. An STNU may be thought of as determining a family of STNs where the contingent links take on each of their possible durations; the individual STNs in the family are called *projections*.

In STNUs, the uncontrolled timepoints are assumed to be either all unobservable or all observable, giving rise to different execution strategies. An STNU is *Strongly Controllable* if there is a single schedule that satisfies the requirements in *all* of the projections. An STNU is said to be *Dynamically Controllable* if there is a strategy for scheduling each executable timepoint that depends only on observations that are available (in the past) at the time it is scheduled. Whether an STNU is Dynamically Controllable or not can be determined by an algorithm that runs in cubic time (Morris 2014). The algorithm tightens some constraints in a way that makes explicit limitations on the execution strategies due to the presence of contingent links.

Some of the tightenings involve a temporal constraint called a *wait*. Given a contingent link AB and another link AC, a wait indicates that execution of the timepoint C is not allowed to proceed until after either B has occurred or some specified amount of time t has elapsed since A occurred. More precisely, it corresponds to the constraint $C - A \geq \min(B - A, t)$. Note that a wait reduces to an ordinary temporal constraint in a projection, since there the value of $B - A$ is fixed.

As mentioned, an STN has an alternative representation as a *distance graph* (Dechter, Meiri, and Pearl 1991). Similarly, there is a representation for an STNU called the *labeled distance graph* (Morris and Muscettola 2005) In the labeled distance graph, each requirement link A $\xrightarrow{[x,y]}$ B is replaced by two edges A $\xrightarrow{y}$ B and A $\xleftarrow{-x}$ B, just as in an STN. For a contingent link A $\xRightarrow{[x,y]}$ B, we have the same two edges A $\xrightarrow{y}$ B and A $\xleftarrow{-x}$ B, but we also have two additional edges of the form A $\xrightarrow{b:x}$ B and A $\xleftarrow{B:-y}$ B. These are called *labeled edges* because of the additional "b:" and "B:" annotations indicating the contingent timepoint B with which they are associated. Note especially the reversal in the roles of x and y in the labeled edges. We refer to A $\xleftarrow{B:-y}$ B and A $\xrightarrow{b:x}$ B as *upper-case* and *lower-case* edges, respectively. Observe that the upper-case labeled weight B:-y gives the value the edge would have in a projection where the contingent link takes on its maximum value, whereas the lower-case labeled weight b:x corresponds to the contingent link minimum value. An upper case edge A $\xleftarrow{B:-t}$ C is also used to represent the wait involving A,B,C considered earlier; it is consistent with the *lower* bound on AC that would occur in a projection where the contingent link has its maximum value.

A POSTNU (Moffitt 2007) is essentially an STNU that has both observable and unobservable (hidden) timepoints. Thus, the controllability problem for a POSTNU may be regarded as a combination of Strong and Dynamic Controllability. Moffitt's algorithm for checking the controllability of a POSTNU is complete but not sound in that it might incorrectly label a POSTNU as controllable. Another algorithm, also relying on the compilation to STNUs is provided by (Bit-Monnot, Ghallab, and Ingrand 2016) that is sound but only complete for a subclass of POSTNUs. A polynomial sound and complete algorithm for assessing the controllability of general POSTNU remains to be found. It is **important to note** one particular point with respect to the semantics. A contingent link may be activated by a hidden timepoint. In that case, if the endpoint is observable, the POSTNU semantics specifies that when it is observed, we learn only the *time* of the endpoint, not the *duration* of the link that was activated by the hidden timepoint. Of course we do learn (or can easily calculate) the time difference between the observed endpoint and any previous known time. Other semantics are possible, and may be useful in some applications, but will not be considered in this paper.

Variable Delay

The Variable Delay STNU (Bhargava, Muise, and Williams 2018) formalism is an STNU extension that relaxes the condition of instantaneous observation of contingent timepoints. In this case, the end of a contingent link is not directly observed; instead after some bounded delay (with upper and lower bounds), it is learned that the contingent timepoint has occurred. The duration of the delay is not observed, so the time at which the contingent timepoint occurred is not directly known. However, bounds on the time of occurrence can be inferred from the other observations.

A Variable Delay STNU may be regarded as a special case of a POSTNU where the original contingent link is chained with a separate contingent link that represents the delayed observation process. The original contingent timepoint is treated as hidden. An example is shown in figure 1. Here XE represents the original contingent link, E is hidden, and EY represents the delayed observation. The link EZ is a requirement that is imposed on the hidden timepoint. In the example, X and Z are executable timepoints and Y is an observable timepoint.

Note that the semantics of Variable Delay STNU implies that Y is a "terminal" timepoint, i.e., the corresponding

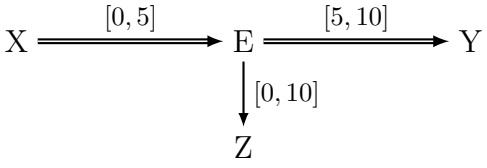


Figure 1: Variable Delay as POSTNU

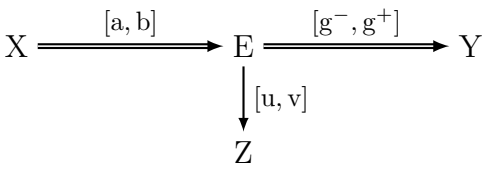


Figure 2: Generic Variable Delay

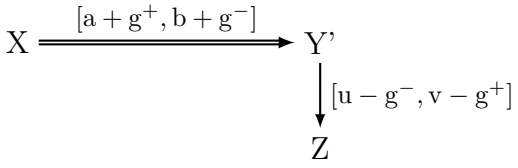


Figure 3: Transformed Variable Delay

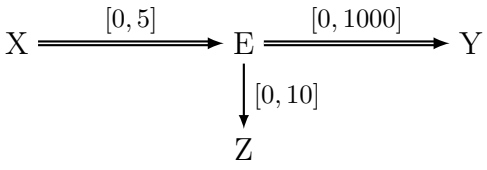


Figure 4: Suboptimal Strategy Example

POSTNU may not impose any requirements on the timepoint representing the delayed observation, and it may not activate a new contingent link. In addition, the delayed observation is "single-headed" in the sense that the POSTNU can have only one contingent link that is activated by the hidden timepoint. In this paper, we will develop solution methods that encompass a wider (though still limited) range of problems.

We now review a somewhat simplified description of the Variable Delay solution procedure (Bhargava, Muise, and Williams 2018), as expressed in terms of a POSTNU, for a generic example (figure 2) that parallels the one used in the Variable Delay paper. For the following discussion, as a notational convenience, we define slack(AB) = q-p for any link AB with bounds [p,q].

The solution procedure starts by checking whether slack(XE) $\leq$ slack(EY). If so, it replaces EY by an infinite delay, which essentially discards the EY observation from the POSTNU.

Otherwise it applies the transformations in table 1, which effectively moves the requirement from the unobservable E to an observable Y' as indicated in figure 3. We have rewritten Y as Y' because, as we will see, it is not really the same timepoint as Y. (More on this later.)

The iterated transformation process converts a Variable Delay problem into one in which timepoints are either unobservable or have zero delays. This is essentially a POSTNU in which all the activation timepoints are observable. These are problems for which dynamic controllability can be checked by previous methods (Bit-Monnot, Ghallab, and Ingrand 2016).

Original edges	Replacement edge
$X \xrightarrow{[a,b]} E \xrightarrow{[p,q]} Y$	$X \xrightarrow{[a+q,b+p]} Y'$
$Z \xleftarrow{[a,b]} E \xrightarrow{[p,q]} Y$	$Z \xleftarrow{[a-p,b-q]} Y'$

Table 1: Variable Delay transformations involving a hidden timepoint E

The Variable Delay paper presents arguments that the transformed problem is dynamically controllable if and only if the original is also, which extends dynamic controllability checking to a wider class of problems.

The paper also presents an execution strategy for the transformed problem. The timepoint designated Y' in figure 3 is treated as though it corresponds to an observation of

$$(t \geq a + g^+) \wedge (Y \vee t \geq b + g^-)$$

where t is the time as measured since X was executed. That is, if Y was observed earlier than time $a + g^+$, then Y' is considered to be observed at time $a + g^+$. If Y is observed between time $a + g^+$ and time $b + g^-$, then Y' is observed when Y is observed. If Y is not observed until after time $b + g^-$, then Y' is considered to be observed at time $b + g^-$. We say Y' is an observable *derived* from Y.

Improved Dynamic Strategy

The Variable Delay paper does correctly determine the dynamic controllability of a Variable Delay problem (as we will later confirm by a different analysis), and it does present a valid dynamic strategy. However, the dynamic strategy obtained is not always the most general possible (in the sense of preserving the greatest flexibility). Consider the example in figure 4. The Variable Delay procedure would essentially discard the Y observation as one that is "highly uncertain," and treat E as totally unobservable. Then, from XE and EZ, we infer an XZ requirement of [5,10]. Notice however that with the original network, if Y is observed at any time in [0,5] we can immediately infer that E has happened, and so it is safe to go ahead with Z. If Y is not observed, we can nevertheless proceed with Z in [5,10]. This is more flexible than [5,10] only.

As another example, suppose the EZ link had bounds [990,1000] instead. Compiling away E would then impose an XZ requirement of [995,1000]. However, if Y has not finished at time 1000, it is nevertheless safe in the original network to hold off on executing Z until Y finishes (since E must still be within the allowed range), which is more flexible and potentially might not happen until 1005 after X.

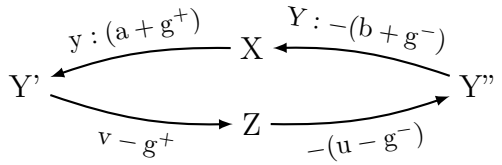
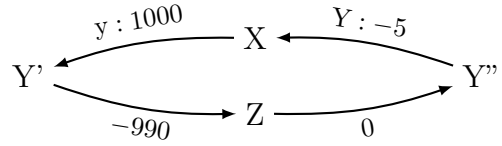


Figure 5: Doubled Y Timepoint

For execution purposes, discarding the Y observation is overly drastic since it can make a contribution to the dynamic strategy even though it is "highly uncertain." However, if the Variable Delay paper did not perform this preliminary step when slack(XE) $<$ slack(EY), then the first transformation in table 1 could produce a paradoxical contingent link where the lower bound is greater than the upper bound. (Note $(\mathrm{b}+\mathrm{g}^-)-(\mathrm{a}+\mathrm{g}^+)=(\mathrm{b}-\mathrm{a})-(\mathrm{g}^+-\mathrm{g}^-)$.)

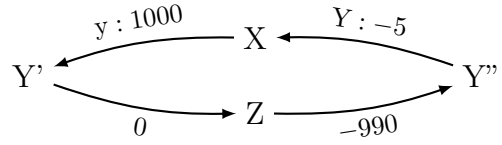
As it turns out, there is an alternative way of resolving this issue that does not require discarding the Y observation. The basic idea is to replace Y by *two* new timepoints Y' and Y", in which we separate the upper and lower bounds. Otherwise, the transformation is essentially the same as in the Variable-Delay paper. Here, we just indicate how this resolves the flexibility issue; later on, we will show how these timepoints arise in a principled analysis.

Figure 5 shows the transformed network as a labeled distance graph. This is semi-reducible if either Y'→Z or Y'→Z→Y" is negative, i.e., if either $\mathrm{v} < \mathrm{g}^+$ or $(\mathrm{v}-\mathrm{u}) < (\mathrm{g}^+-\mathrm{g}^-)$. We then have a semi-reducible negative cycle if the whole cycle is negative, i.e., if $(\mathrm{v}-\mathrm{u}) < (\mathrm{b}-\mathrm{a})$. (The g^+ and g^- terms cancel.) This gives the same gross determination of dynamic controllability as the previous (Bhargava, Muise, and Williams 2018) procedure but differs in terms of the specific dynamic strategy. For the example, we get



and then, applying the usual STNU reductions, we end up with $\mathrm{X} \xrightarrow{10} \mathrm{Z}$ and $\mathrm{X} \xleftarrow{\mathrm{Y}:-5} \mathrm{Z}$ edges, which corresponds to a dynamic strategy of "Wait for Y until time 5 after X, and then execute Z before time 10 after X," which is the more flexible strategy we discussed earlier.

For the example where EZ has bounds [990,1000], the Y'Z and ZY" bounds are the only ones affected, and we get the situation depicted in the following figure.



Here, Y" observes the "Wait for Y until time 5 after X" condition, and then Z is released 990 units later. We will see later that the y:1000 bound on XY' can be interpreted as an upper bound of "Y or 1000 after X, whichever comes later," and then the same upper bound applies to Z. This strategy also matches our intuition.

These examples underscore our understanding that Y' and Y" are NOT the same timepoint as Y, although they are derived from it. The original Y timepoint in figure 2 has bounds of $[\mathrm{a}+\mathrm{g}^-, \mathrm{b}+\mathrm{g}^+]$ and these would need to be used, for example, if we were to consider placing requirements on Y itself. (This is apparently not within the scope of the Variable Delay formalism.)

In this section, to facilitate comparisons, we have used variable names that approximate those used in the Variable Delay paper. However, from now on we will adopt the convention, in most cases, of using bounds $[q^-, q^+]$ for any link whose endpoint is Q. We hope this will be useful as a mnemonic aid.

Single-Headed POSTNUs

The hidden timepoints in a POSTNU may be partitioned into separate groups whose elements are connected to each other by contingent links. A group is thus a connected component of the undirected graph obtained by *(i)* removing all requirement links from the POSTNU and *(ii)* replacing contingent links by their undirected variant. Since the STNU definition does not permit two contingent links to have the same endpoint, each group, together with an activation timepoint, will form a tree-like structure.

We now turn our attention to the special case where the hidden timepoints occur in groups consisting of linear chains of contingent links with a single non-hidden entrance and single non-hidden exit. We will call these *Single-Headed* POSTNUs.

For instance in a network $A \Rightarrow E_1 \Rightarrow E_2 \Rightarrow B$, E_1 and E_2 are hidden timepoints that belong to the same hidden group. (Notation convention: any E_i timepoint is hidden.)

Without loss of generality we may assume the entrance timepoint is controllable since otherwise it could be replaced by a controllable with a $[0, 0]$ link to the original entrance. The exit timepoint is necessarily observable.

In this paper, **we exclude direct requirement links between two hidden timepoints**,[1] but otherwise the hidden timepoints (and entrance and exit) may participate in requirement links to other timepoints in the network. We then assume without loss of generality that timepoints directly linked to hidden timepoints are controllable, using $[0, 0]$ link replacement if necessary.

As shown in figure 2, the Variable Delay problems may be regarded as a special case of Single-Headed POSTNUs, with limitations on the hidden groups and requirement links.

Analysis From First Principles

In our analysis we will first restrict our attention to simple Single-Headed POSTNUs, where the hidden groups each contain only one hidden timepoint, and later relax that restriction.

In an earlier section, we described a "doubling" strategy that enhanced the flexibility of execution. We now present

[1] For simplicity—the consequence of allowing them is unclear.

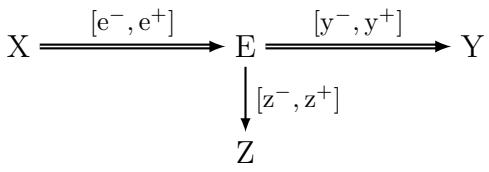


Figure 6: Generic Simple Problem

a first principles analysis in which the doubling arises naturally. The analysis focuses on mathematical equivalences that are independent of context. This eliminates some of the contextual restrictions that applied in the Variable Delay setting. As a side-benefit, the analysis sheds some additional light on the semantics of the upper-case and lower-case labeled edges used in the STNU work (Morris 2014).

For a POSTNU, we may divide the projections into groups that have the same values for their observable timepoints. We will call these groups *macro-projections*. The full projections that also specify the hidden timepoint values will be called *micro-projections*. Thus, each macro-projection consists of a set of micro-projections. In effect, each macro-projection, considered in isolation, may be regarded as a separate Strong Controllability problem whose projections are its micro-projections. Then each hidden timepoint will have a range of values within a particular macro-projection, and this range will depend on the values of the observables in the macro-projection.

For example, with the POSTNU (where E is hidden)

$$\text{X} \xrightarrow{[0,10]} \text{E} \xrightarrow{[0,10]} \text{Y}$$

the macro-projection where XY = 15 consists of all the micro-projections where XE and EY sum to 15, such as $6 + 9$, $10 + 5$, etc. Within this set of micro-projections, E can range from 5 to 10 (after X). Notice while E can vary, the lower and upper bounds of the range, E_{lo} and E_{hi}, are fixed within the macro-projection. As we will see, their values can be expressed in terms of formulas involving the observables. Thus, we may regard them as virtual timepoints that live within the macro-projection, or *virtual observables* (although we only know their values *after* the relevant real observables have been observed). If we now impose a $[z^-, z^+]$ requirement on EZ, where Z is an executable timepoint, a worst-case analysis suggests we should enforce that by adding constraints $\text{Z} \geq \text{E}_{\text{hi}} + z^-$ and $\text{Z} \leq \text{E}_{\text{lo}} + z^+$. In contrast to the case for virtual observables, for the *executable* timepoint Z, we *do* need to know these constraints are satisfied by the time Z is scheduled.

Redirected Requirements We now consider this analysis in more detail for the simple generic problem shown in figure 6 (similar to Variable Delay). Here, X and Y are non-hidden timepoints. Both of these give us information bounding the occurrence of E as follows:

$$\begin{array}{ll} \text{E-X} & \geq e^- \\ \text{E-X} & = \text{(Y-X)-(Y-E)} \\ & \geq \text{(Y-X)} - y^+ \end{array}$$

Thus,

$$\text{E-X} \geq \max(e^-, \text{(Y-X)} - y^+)$$

Similarly,

$$\begin{array}{ll} \text{E-X} & \leq e^+ \\ \text{E-X} & = \text{(Y-X)} - \text{(Y-E)} \\ & \leq \text{(Y-X)} - y^- \end{array}$$

so

$$\text{E-X} \leq \min(e^+, \text{(Y-X)} - y^-)$$

It is not hard to see that these are *tight* bounds; they represent the minimum and maximum extent of E-X within the macro-projection determined by X and Y. As discussed in the example, we will designate the lower and upper virtual observables by E_{lo} and E_{hi} respectively.

It is convenient to simplify the formulas by writing $\dot{Y}$ for Y-X. Thus, the X to E link has inferred bounds of

$$[\max(e^-, \dot{Y} - y^+), \min(e^+, \dot{Y} - y^-)]$$

Given a particular macro-projection, a dynamic strategy will need to specify a value for Z that works for *all* the associated micro-projections, i.e., for all the values of E within this range. Thus, we require $\text{Z} - \text{E} \geq z^-$ for each such E. It is not hard to see [2] that this is true **if and only if** it is true for the upper bound of the range, i.e., $\text{Z} - \text{E}_{\text{hi}} \geq z^-$. Similarly, the lower-bound requirement is **equivalent** to $\text{Z} - \text{E}_{\text{lo}} \leq z^+$.

We can rewrite these requirements as supplying a lower bound for XZ of $\text{E}_{\text{hi}} + z^-$ or

$$\min(e^+ + z^-, \dot{Y} + z^- - y^-)$$

and an upper bound of $\text{E}_{\text{lo}} + z^+$ or

$$\max(e^- + z^+, \dot{Y} + z^+ - y^+)$$

Notice the min/max modifiers have become reversed with respect to the lower and upper bounds. One consequence is that the bounds now represent implicit disjunctions rather than implicit conjunctions. However, we will see that the two alternatives can be processed together in a way that avoids an exponential blowup.

Observability Tightening These derived bounds may not be directly observable. For example, $(z^- - y^-)$ may be negative in which case the value of $\dot{Y} + (z^- - y^-)$ is unknown until the later time when (Y-X) is actually observed. If $(z^- - y^-)$ is non-negative, then $\dot{Y} + (z^- - y^-)$ is observable and can be left unchanged. Otherwise, when executing Z we must replace $\dot{Y} + (z^- - y^-)$ by the observable $\dot{Y}$, which gives a strictly tighter lower bound that guarantees the actual bound will be satisfied. We call this process *observability tightening*. It is important to note that we only apply it to *executable* timepoints, which is where the dynamic strategy applies.

When executing Z, the upper-bounds also need "observability tightening" but the process is different because of the asymetry of observation with respect to time. For example, if $(z^+ - y^+)$ is negative, then the strictly tighter bound derived

[2] $\text{Z} - \text{E} \geq z^-$ for each E, implies $\text{Z} - \text{E}_{\text{hi}} \geq z^-$. Conversely, if $\text{Z} - \text{E}_{\text{hi}} \geq z^-$ then $\text{Z} - \text{E} \geq \text{Z} - \text{E}_{\text{hi}} \geq z^-$ for each E.

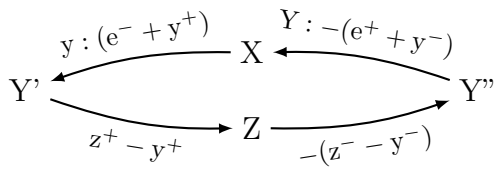


Figure 7: Distance Graph

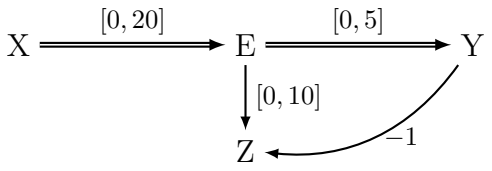


Figure 8: Cross Requirement Example

from $\dot{Y} + (z^+ - y^+)$ is "minus infinity", which is equivalent to dropping the $\dot{Y} + (z^+ - y^+)$ term from the max expression. If $(z^+ - y^+)$ is non-negative, then the term can be left unchanged.

Derived Observables Rather than interpreting the bounds on Z directly, we will pursue an alternative approach here, and decompose them by introducing intermediates with respect to the $\mathrm{E_{lo}}$ and $\mathrm{E_{hi}}$ values. Although $\mathrm{E_{lo}}$ and $\mathrm{E_{hi}}$ are only *virtual* observables whose values may not be known until later, we can form real observables from them by adding approriate delay terms. For example, $\mathrm{E_{hi}} + y^- = \min(e^+ + y^-, \dot{Y})$ corresponds to an observation of "Y or $e^+ + y^-$ after X, whichever is earlier," and $\mathrm{E_{lo}} + y^+ = \max(e^- + y^+, \dot{Y})$ may be paraphrased as "Y or $e^- + y^+$ after X, whichever is later." We will designate these derived observables as Y" and Y', respectively.

This motivates us to expand the lower bound for Z as

$$\min(e^+ + y^-, \dot{Y}) + (z^- - y^-)$$

and the upper bound as

$$\max(e^- + y^+, \dot{Y}) + (z^+ - y^+).$$

We will identify $-\min(u, \dot{Y})$ with the upper-case labeled weight Y:$-u$ and $\max(v, \dot{Y})$ with the lower-case labeled weight y:v, as used in an STNU labeled distance graph. (Morris 2014). (This will be justified later, but note that semantically, $\min(u, \dot{Y})$ is the same as the *Wait for Y until u after X* condition in an STNU.)

Introduction of the intermediate Y' and Y" thus produces the labeled distance graph shown in figure 7. This may be compared with figure 5.

Example We reiterate that the Y timepoint is distinct from the added Y' and Y" timepoints. The correlation between them is captured by the labeled weights in the distance graph. Consider, for example, the network shown in figure 8. If the YZ edge was not there, the network would be Dynamically Controllable, since Z could be executed between 0 and 5 after Y is observed. However, the YZ edge prevents that strategy by requiring Z to come before Y, so the full network is not Dynamically Controllable. The distance graph after the transformations is shown in figure 9. Notice the Lower-Case reduction applied to XYZ produces an XZ edge of weight -1, which then forms a semi-reduced negative cycle with the ZY"X path.

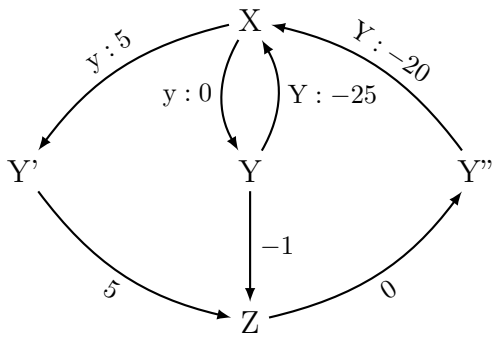


Figure 9: Cross Distance Graph

Hidden Timepoint Elimination After the requirement edges between Z and E are replaced by the corresponding edges between Z and X, E will be free of "side" links. At that point, the XE link can be composed with the EY link, giving a combined contingent link of XY with bounds $[e^- + y^-, e^+ + y^+]$, and E can be eliminated.

Now we return to the general case where there is a chain of hidden timepoints

$$\mathrm{X} \Rightarrow \mathrm{E_1} \Rightarrow \ldots \Rightarrow \mathrm{E_n} \Rightarrow \mathrm{Y}$$

between X and Y. Consider the first timepoint $\mathrm{E_1}$. The analysis that produced $\mathrm{E_{lo}}$ and $\mathrm{E_{hi}}$ depended only on knowing the contingent bounds for XE and EY. Viewing $\mathrm{E_1}$ as if it were E, we know the bounds for XE directly, and we can compute bounds for EY by composing the contingent links in the $\mathrm{E_1} \Rightarrow \ldots \Rightarrow \mathrm{Y}$ path. [3] We can then proceed as in the single E case to eliminate $\mathrm{E_1}$. This process can be repeated with the other hidden timepoints in the chain until they are all eliminated.

At this point, what remains is a labeled distance graph with no hidden timepoints, which is a form suitable for input to a standard cubic Dynamic Controllability checking algorithm for STNUs (Morris 2014). This leads us to the following theorem.

Theorem 1 *For the given class of Single-Head POSTNUs, the transformation process followed by the standard cubic Dynamic Controllability checking algorithm provides a complete decision procedure.*

Proof: We have seen that the first transformation step replaces the original requirement constraints with equivalent ones. Because of the equivalence, this necessarily leaves the set of valid dynamic strategies unchanged. The observability tightening step does restrict the network, but any strategies eliminated by the step would be non-dynamic since they

[3] Requirements do not affect the domains of contingent links.

would depend on unobserved values. Thus, the set of dynamic strategies before and after the transformation process is the same. (This may be empty if the network is not dynamically controllable.)

Next we justify the identification of the max/min expressions with the labeled weights by showing they behave the same with respect to the key reductions used by the Dynamic Controllability checking algorithm. In the following, we assume $u > 0$ and $v \geq 0$ and $W \neq Y$.

Upper-Case Reduction

$$\begin{aligned} -\min(u, \dot{Y}) + v &= -\min(u - v, \dot{Y} - v) \\ &= -\min(u - v, \dot{Y}) \end{aligned}$$

Lower-Case Reduction

$$\begin{aligned} -u + \max(v, \dot{Y}) &= \max(v - u, \dot{Y} - u) \\ &= v - u \end{aligned}$$

Cross-Case Reduction

$$\begin{aligned} -\min(u, \dot{W}) + \max(v, \dot{Y}) &= v - \min(u, \dot{W}) \\ &= -\min(u - v, \dot{W}) \end{aligned}$$

Label Removal

$$-\min(-v, \dot{Y}) = v$$

Note the use of the applicable **observability tightening** in the Upper and Lower cases. The Cross-Case reduction applies the Lower and Upper derivations in succession. [4] Label Removal follows from min simplification since $\dot{Y} \geq 0$.

The theorem then follows from the completeness of the (Morris 2014) algorithm. □

Note that this result extends and unifies the previous classes of POSTNU for which complete and tractable decision procedures are known (Bit-Monnot, Ghallab, and Ingrand 2016; Bhargava, Muise, and Williams 2018).

Multi-Headed Observations

In the previous sections, we discussed problems where bounds on the occurrence of a hidden timepoint could be inferred from a single observation. In this section, we consider the combined effect of multiple relevant observations, the so-called "Multi-Headed Problem," where there can be multiple determinations that a hidden event has occurred, each of which has its own bounds. This is illustrated in figure 10 for a two-headed problem.

In the figure, X and Z are executable timepoints, while E is hidden, and Y and W are observables. The past occurrence of E can be inferred from an observation of either Y or W, which also provide (different) bounds on when E occurred.

In this section, we present some partial results concerning these kinds of problems, and some possible approaches. A complete solution remains a challenge for future work.

Local Dynamic Controllability

Although the ultimate goal is to develop transformation methods that apply to POSTNU fragments independent of context, a useful first step is to characterize Dynamic Controllability for certain fragments in isolation. These can be used as a check on the validity of more general approaches,

[4] Hint: first apply the LC derivation using $u' = \min(u, \dot{W})$.

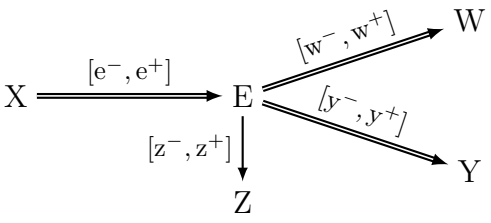


Figure 10: Two-Headed Problem

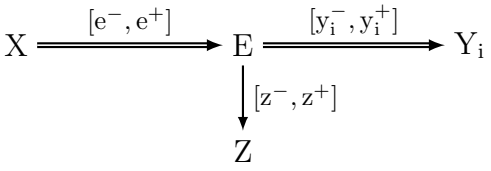


Figure 11: Multi-Headed Problem

and thus help to guide further research. Here we prove a result of this kind. It also illustrates some distinctions between the problem of checking Dynamic Controllability, and aspects regarding flexibility of execution.

The theorem applies to a problem with any number of "heads." Consider the generic example in figure 11 where i is repeated from 1 to n. Here X and Z are executable timepoints, while E is hidden, and the Y_i are separate observables.

Theorem 2 *The network in figure 11 is Dynamically Controllable if and only if* **either** *slack(EZ) ≥ slack(XE)* **or** *slack(EZ) ≥ slack(EY$_i$) for some i such that $z^+ \geq y_i^+$.*

Proof: If slack(EZ) ≥ slack(XE), then $z^+ + e^- \geq z^- + e^+$ and then executing Z in a way that satisfies XZ = $[z^- + e^+, z^+ + e^-]$ constitutes a dynamic strategy since the EZ requirement will be satisfied no matter the outcome of the XE contingent link. Note that XZ can be placed as a requirement even if $z^+ + e^-$ and $z^- + e^+$ are negative since X is controllable.

Also, if slack(EZ) ≥ slack(EY$_i$) and $z^+ \geq y_i^+$ for some i, then $z^+ - y_i^+ \geq z^- - y_i^-$, and executing Z in a way that satisfies Y_i Z = $[z^- - y_i^-, z^+ - y_i^+]$ will constitute a dynamic strategy since then the EZ requirement will be satisfied no matter the outcome of the EY$_i$ contingent link. Note Z can be scheduled to satisfy this Y_iZ since $z^+ \geq y_i^+$ and Y_i is observable.

Thus, the "if" direction is satisfied.

Conversely, suppose slack(EZ) < slack(XE) and, for all i, either slack(EZ) < slack(E Y_i) or $z^+ < y_i^+$.

Let Q be the set of i such that slack(EZ) < slack(E Y_i). Define q = min {slack(E Y_i) : i in Q}. Then slack(EZ) < q.

We define two projections P1 and P2 as follows.

- In P1, XE has its maximum extent e^+
- In P2, XE has extent $e^+ - q$
- For $i \in$ Q:
  - In P1, EY_i has its minimum extent y_i^-

– In P2, EY_i has extent $y_i^- + q$

- For $i \notin$ Q:
  – For both P1 and P2, Y_i has its maximum extent y_i^+

Note that for $i \in$ in Q, the observed XY_i = XE + EY_i has the same extent $e^+ + y_i^-$ in both P1 and P2.

Also note that for $i \notin$ Q, $z^+ < y_i^+$, so in both P1 and P2, none of the Y_i will have been observed by the time Z reaches its upper bound, and must have been scheduled.

From the above we see that P1 and P2 cannot be distinguished by any dynamic strategy, so Z must be scheduled at the same time in both projections.

Finally, we note that the value of E in P1 and P2 differs by an amount q. But slack(EZ) $<$ q and Z is fixed. It follows that the EZ constraint must be violated in either P1 or P2, which contradicts the assumption of a dynamic strategy. Thus the network is not Dynamically Controllable, proving the "only if" direction. □

We remark that Theorem 2 is consistent with the Variable Delay transformations, as well as the doubling strategy, with respect to the determination of Dynamic Controllability. A point of interest is that for determining Dynamic Controllability, the property of importance is the existence of at least one "head" (or activation "tail") with less slack (i.e., uncertainty) than the requirement. However, we have seen that flexibility of execution can be enhanced by opportunistic use of observations whose slack may exceed that of the requirement.

Global Dynamic Strategy

In this section, we explore a first principles approach similar to that used in the single-headed case, and see what additional issues arise. In particular, we consider the two-headed problem in figure 10.

In this two-headed example, the observables are X, Y, and W. Each of the observations gives us information bounding the occurrence of E. We can then derive overall bounds in a manner similar to that used in the single-head case. This results in the following inferred bounds for the X to E link. (Recall that $\dot{Y}$ abbreviates Y-X, and $\dot{W}$ abbreviates W-X.)

$$[\max(e^-, \dot{Y} - y^+, \dot{W} - w^+), \min(e^+, \dot{Y} - y^-, \dot{W} - w^-)]$$

As for the single-head case, we define virtual observables E_{lo} and E_{hi} in terms of these bounds, and add constraints $Z \geq E_{hi} + z^-$ and $Z \leq E_{lo} + z^+$ to give an X to Z link with lower bound

$$\min\{e^+ + z^-, \dot{Y} + (z^- - y^-), \dot{W} + (z^- - w^-)\}$$

and upper bound

$$\max\{e^- + z^+, \dot{Y} + (z^+ - y^+), \dot{W} + (z^+ - w^+)\}.$$

Observability tightening must again be applied since Z is an executable timepoint. One difference from the single-head case is that there are two observables in each bound, and the tightening needed may be different in each case. In particular, the upper-bound tightening (which potentially drops terms from the max expression) may eliminate one or both of the observable terms. If it eliminates both, this leaves a single value, which is analogous to an application of the Lower Case Reduction. If it eliminates only one, this leaves what is effectively a single-head expression. It may also drop no terms, leaving an expression with multiple observable terms.

At this point it is unclear how far the analogy to the single-head case can be carried further. Considering just the lower-bound expression, the values $(z^- - y^-)$ and $(z^- - w^-)$ added to the Y and W observables may be different, so there is no one term that we can "take outside," leaving a "bare" observable, as we did for the single-headed case. This makes the approach of introducing intermediate observables unclear, and even if we did, the double-observable expressions cannot be identified with conventional STNU labels. [5]

However, if we could sidestep the problem of checking Dynamic Controllability, the multi-observation bounds on executable timepoints could in fact be interpreted in accordance with a dynamic strategy. For example, consider a lower bound of $\min(10, \dot{Y} + 5, \dot{W})$. (Note that after observability tightening, any quantities added to an observable will be non-negative.) This can be interpreted as an observation of "Y+5 or W or 5 after X, whichever is earlier." Similarly, an upper bound of $\max(20, \dot{Y}, \dot{W} + 10)$ corresponds to "Y or W+10 or 20 after X, whichever is later."

Closing Remarks

We have built on previous work in the area of STNUs, especially Variable Delay, and extended it to a POSTNU setting. By means of a detailed First Principles analysis, we have shown how to achieve a more flexible dynamic strategy for execution. The results provide for additional context in terms of network configuration. For the "single-headed" class of problems considered, the determination of Dynamic Controllability is complete and correct, and the dynamic strategy preserves the full flexibility. We have also explored multi-headed problems and presented partial results in this area.

Since the Dynamic Controllability and Strong Controllability problems for STNUs are tractable, and since POSTNUs are essentially a combination of the two, it is plausible to think that the general POSTNU problem might be tractable, although a general solution to this problem is unknown at the present time. It seems to be a difficult problem to analyze, but a very interesting one, in view of the "arrow of time" with respect to the observables, but not the unobservables. The Variable Delay paper may be regarded as establishing a beachhead in terms of new approaches to this problem, and the current paper makes further forays in this area. We are hopeful that future advances may lead to the sought-after general solution.

References

Bhargava, N.; Muise, C.; and Williams, B. 2018. Variable-delay controllability. In *International Joint Conference on Artificial Intelligence (IJCAI'18)*.

[5] Of course, one could introduce an explicit disjunction and handle the observables separately, but that seems to abandon the search for a tractable algorithm.

Bit-Monnot; Ghallab, M.; and Ingrand, F. 2016. Which contingent events to observe for the dynamic controllability of a plan. In *International Joint Conference on Artificial Intelligence (IJCAI'16)*.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence* 49:61–95.

Hunsberger, L. 2009. Fixing the semantics for dynamic controllability and providing a more practical characterization of dynamic execution strategies. In *International Symposium on Temporal Representation and Reasoning (TIME'09)*.

Moffitt, M. D. 2007. On the partial observability of temporal uncertainty. In *AAAI Conference on Artificial Intelligence (AAAI'07)*.

Morris, P., and Muscettola, N. 2005. Dynamic controllability revisited. In *AAAI Conference on Artificial Intelligence (AAAI'05)*.

Morris, P.; Muscettola, N.; and Vidal, T. 2001. Dynamic control of plans with temporal uncertainty. In *International Joint Conference on Artificial Intelligence (IJCAI'01)*.

Morris, P. 2014. Dynamic controllability and dispatchability relationships. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR'14)*.

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

Executing Contingent Plans: Addressing Challenges in Deploying Artificial Agents

Christian Muise
christian.muise@ibm.com
IBM Research

Miroslav Vodolán
MVodolan@cz.ibm.com
IBM Watson

Shubham Agarwal
Shubham.Agarwal@ibm.com
IBM Research

Ondrej Bajgar
ondrej@bajgar.org
Future of Humanity Institute, University of Oxford
(*work done while at IBM Watson*)

Luis Lastras
lastrasl@us.ibm.com
IBM Research

Josef Ondrej
josef.ondrej@ibm.com
IBM Watson

Abstract

The vast majority of research in automated planning focuses on generating a plan from an initial problem specification; from the theoretical properties of this task to the implementation details required to do so efficiently. While such work is often motivated by practical applications, there is far less understanding of the issues associated with executing plans in online environments. In this work we focus on this understudied area, and the challenges / opportunities that arise when executing complex plans. Unlike many works in plan execution, we consider a form of contingent plans as the source for execution; their complexity stems from the sophisticated representation of the action effects used to model the uncertainty in the world. The key contribution of our work is a proposed executor that can reason using the sophisticated action effects, and we demonstrate the impact this can have empirically. In support of an effective executor, we also consider (1) the connection between the execution context and the planner's view of the state of the world; and (2) the separation between the execution of an action (the aspect that affects the outside environment) and the realization of its effects (the aspect that captures what has actually changed).

1 Introduction

The field of automated planning spans the entire pipeline of developing and deploying a cognitive agent that needs to interact with an environment. From the acquisition of planning models to the online execution of plans, there is a broad set of challenges that must be addressed. By far, most of the focus of the planning research community is directed towards the *generation* of plans given a model. The plans themselves are the end-game of this research. We focus on the crucial and understudied aspect of *executing complex plans*, and the challenges that come with it (we use *complex* to refer to the fact that we use contingent plans with a sophisticated representation of action effects for modeling uncertainty). The added complexity provides a rich setting for the domain modeler to capture the world effectively, and further allows for more efficient implementations.

There is a variety of existing work on plan execution and execution monitoring, and we discuss its relation to our work later in Section 6. However, our work is distinct by its focus on deploying rich plans that deal with uncertainty; i.e., the challenges and opportunities that arise from execution monitoring in a setting of modeled non-determinism. A high-level

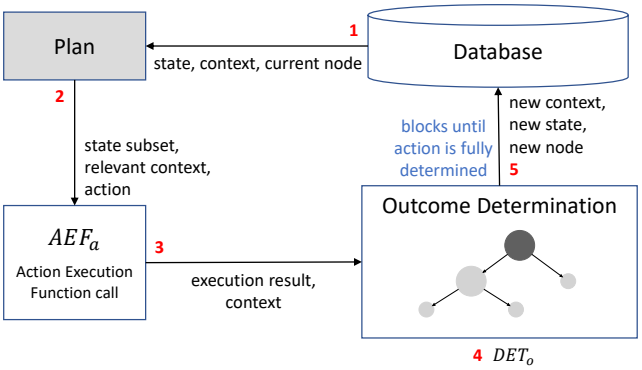


Figure 1: Architecture for executing contingent plans.

view of our proposed execution architecture can be seen in Figure 1, and we highlight the key aspects here with further details later in Section 3.

We assume that plans take the form of a controller or policy network, a format used by existing planners that deal with uncertainty (Geffner and Geffner 2018; Muise, Belle, and McIlraith 2014), but additionally address the key specification details external to the planning process that is required for execution. Effective planning relies on identifying the right level of abstraction for a target system, and we treat this abstraction as a first-class citizen in our work. We define a crisp description of the connection between the planning model and the context that is maintained as part of the real-world our agent executes in (referred to as state and context respectively in Figure 1).

Following a clear mapping between the state and context, we address the interplay between the execution of an action and the realization of its effect on the world (points 3 and 4 in Figure 1). This is particularly interesting when an action can change the world in a non-deterministic manner. For this work, we assume that the effects of an action, albeit non-deterministic, are fully observable and accurate. In practice, we have found using a fully observable setting not to be a limitation (as opposed to partial observability), as many uncertain settings can be adequately modeled with pre-existing low order compilation techniques, e.g., using K_1 compilations as in (Palacios and Geffner 2009) or *width-1* compilations proposed by (Bonet and Geffner 2014).

As for the accuracy of effect determination, this is a simplifying assumption that we make in order to maintain clarity of the current work. It is a separate and important issue to address the detection and handling of inaccurate or noisy sensing during execution, and we defer these details to future work on the executor we present in this paper. We further assume that action execution is blocking and non-concurrent (i.e., we do not yet extend to the general setting of contingent temporal plan execution – an area that is quite understudied, even theoretically).

The largest contribution of our work is to focus on the execution of complex non-deterministic effects. With a complex model of uncertainty in how the state of the world can change as the result of an action, we must consider how best to embed this complexity in an executing agent (step 4 in Figure 1). We formalize the complex nesting of action effects, address some of the challenges in improving the efficiency of realizing these effects, and empirically demonstrate the potential of sophisticated effect determination.

After providing the necessary background in Section 2, in Section 3 we detail our three main contributions in executing contingent plans: (1) the connection between the execution context and the planner's representation of the state of the world; (2) the separation between the execution of an action (the aspect that can actually affect the environment) and the realization of its effects (the aspect that captures what has actually changed); and (3) the interplay between components of a complex action effect. In Section 4 we present an evaluation of the impact our proposed approach to action determination can have on the efficiency of execution. In Section 5 we draw the connection between our work and traditional execution monitors. We follow with a discussion of related work in Section 6 and conclude in Section 7.

2 Preliminaries

The style of plans we consider are contingent plans that would be generated from a Fully Observable Non-Deterministic (FOND) or Fully Observable Probabilistic (FOP) planner (Muise, McIlraith, and Beck 2012; Camacho, Muise, and McIlraith 2016). We do not go into details on how these plans are produced, but instead focus on the planning problem and solution specifications (the latter being a core component of the input for execution). Unlike traditional planning specifications, we assume additional information is available to specify how actions are executed and observed (using callback functions described below).

Here, we review some of the commonly used notation, and extend it to the rich setting required for effective execution.[1] For simplicity, we will assume a FOND setting, but all of the techniques work equally well in the probabilistic setting as well: the key difference between the FOND and probabilistic setting lays with how the plans are produced, as the solution form is the same. The execution of a probabilistic plan need not take into account the probabilities assigned to action effects, and so we do not consider it further.

[1]For those familiar with non-deterministic formalisms, we do not assume fairness of the domain model. This, however, is irrelevant for the purpose of *executing* plans.

Definition 1 (Planning Problem)**.** A FOND planning problem $\langle \mathcal{F}, \mathcal{I}, \mathcal{A}, \mathcal{G} \rangle$ consists of fluents $\mathcal{F}$ that describe what is true or false in the world, an initial state $\mathcal{I} \subseteq \mathcal{F}$ where the planning agent begins execution, a set of actions $\mathcal{A}$ that the agent can do, and a goal $\mathcal{G} \subseteq \mathcal{F}$ that specifies the partial assignment of fluents that must be achieved. A *complete state* (or just *state*) is a subset of the fluents $\mathcal{F}$ that are presumed to be true (all other presumed to be false); a *partial state* is similarly defined but without any presumption about the truth of fluents outside of the set.

Generally, we adopt the standard notation for FOND planning. The one exception is that we do *not* make the simplifying assumption that the action effects are just a set of one or more non-deterministic effects, one of which will be chosen during execution, as is usually assumed in FOND planning. In practice, the action effects are a nesting of *and* and *oneof* clauses (the latter referring to the notion that *exactly one* of the sub-clauses must be used).

We retain this complexity for two key reasons: (1) the arbitrary nesting gives us a level of sophistication that the plan executor can tap into (demonstrated in Section 3.3 and the evaluation later); and (2) it provides a far more natural form of action specification for the modeler to work with during design and maintenance of the planning model.

Definition 2 (Complex Actions)**.** Let $a \in \mathcal{A}$ be an action. $\mathrm{Pre}_a \subseteq \mathcal{F}$ then denotes the *precondition* of a – the set of fluents that must hold for a to be applicable (that means a is applicable in state s iff $\mathrm{Pre}_a \subseteq s$). Eff_a denotes the effect formula of a. An effect (sub)formula is one of the following:

$(\neg)f$: a fluent $f \in \mathcal{F}$ or its negation
$\bigwedge_\varphi \varphi$: a conjunction of effect subformulae
$\bigoplus_\varphi \varphi$: a mutually exclusive disjunction of effect subformulae (i.e., exactly one is chosen)

The effect of an action can be viewed as an *and-or* tree. A *realization* of the effect consists of all fluents or their negations that appear in the sub-tree which includes exactly one child of each *or* node (here, we refer to them as *oneof* nodes) and all children of each *and* node. Such realization can be thought of as one possible result of the action's execution. We use $\mathcal{R}(a)$ to refer to the set of all realizations for action a, and for each realization $r \in \mathcal{R}(a)$, we define DEL_r to be the set of fluents removed from the state as a result of this action and ADD_r the fluents to be added. We make the non-standard (but non-restricting) assumption that for all actions and their realizations, $ADD_r \cap DEL_r = \emptyset$.

The arbitrary nesting of fluents, $\bigwedge$, and $\bigoplus$ operators mirror the common description of FOND problems in PDDL using `and` and `oneof` clauses (Geffner and Bonet 2013). For execution purposes, we assume that every sub-formula of an action's effect is uniquely identifiable, and will use set notation to refer to those components (e.g., $\varphi \in \mathrm{Eff}_a$). Note that we only allow negation at the leaf level of the effect, rather than allow arbitrary negation of sub-formulae. This is due to the fact that the negation of $\bigoplus$ clauses is ill-defined from the planning perspective, as is the use of *or* in effects, which would arise from negating an *and* clause (this is in contrast with the arbitrary nesting typically allowed in ac-

tion preconditions in the literature).

Solutions to a FOND problem come in two main flavours: (1) policies mapping each state of the world to an action, and (2) controllers, where the nodes and edges respectively correspond to actions and possible outcomes (Geffner and Bonet 2013). We adopt the latter in this work.

Definition 3 (Contingent Plan)**.** A solution to a FOND planning problem (or *contingent plan*) is a graph $\langle \mathcal{N}, \mathcal{E}, n_0 \rangle$, where $\mathcal{N}$ are the nodes of the graph corresponding to the actions the agent should take and $\mathcal{E}$ are the edges corresponding to the possible outcomes of each action associated with the nodes. We use $n_0 \in \mathcal{N}$ to refer to the initial node in which the agent should begin executing. We further assume that we have a function mapping nodes to actions ($action : \mathcal{N} \to \mathcal{A}$) and functions mapping the realizations to the successors of a node ($next_n : \mathcal{R}(action(n)) \to successors(n)$).

The extra notation for mapping nodes and edges to the original FOND problem allows us to tie together the generated plan and its execution. We make no assumptions on the embodiment of the executing agent, but assume that black-box callback functions are available for (1) the initial execution of the action (which affects the world) and (2) the realization of that action's impact.

Definition 4 (Callback Functions)**.** We define the *action execution function* AEF_a to be the function used to execute planning action $a \in \mathcal{A}$. We define the *determiner* DET_o to be the function that is used to determine which outcome of the non-deterministic effect $o = \bigoplus_{\varphi} \varphi$ has occurred.

3 Approach

Our aim is to provide a coherent and effective strategy for deploying agents that are based on the execution of a contingent plan. In this section, we identify some of the key challenges that arise, and propose a solution to each of them.

The high-level architecture we propose for executing contingent plans can be found in Figure 1. The execution involves the following phases:

1. At every iteration of the execution monitor, we have the state of the world (planner view), context of variable assignments (execution view), and the plan's current node n along with the corresponding action $a = action(n)$.
2. The executor retrieves the relevant context and state for action a (Section 3.2).
3. The AEF_a function is called with the filtered context.
4. The action's effect is determined (Section 3.3).
5. State and context are updated (Section 3.3).

To motivate the type of complex actions we wish to execute, consider a home assistant scenario where the virtual agent can perform common tasks around the household. Figure 2 shows an example PDDL representation of an action to prepare the garage for a car to exit, and Figure 3 shows the accompanying effect structure. Notice that some of the non-determinism is independent (e.g., the locked status of the garage door and alarm status), and some of the non-determinism contains a dependency (e.g., the open status only plays a role when the door is not locked). We address these points further in the following sections.

```
(:action prepare_garage_car_exit
  ; garage door is locked and closed,
  ;  and garage lights are off
  :precondition (and (garage_door_locked)
                     (not (garage_door_open))
                     (not (garage_lights_on)))
  :effect (and
            ; garage lights are turned on
            (garage_lights_on)
            (oneof
              ; a visual sensor in garage sets
              ;  off the alarm if car is gone
              (garage_alarm_on)
              (and
                ; the car is inside the garage
                (not (garage_alarm_on))
                (have_fuel_level)))
            (oneof
              ; garage door is malfunctioning
              (garage_door_locked)
              (and
                ; door unlocks successfully
                (not (garage_door_locked))
                (oneof
                  ; garage door is obstructed
                  ;  by some object
                  (not (garage_door_open))
                  ; garage door opens
                  (garage_door_open))))))
```

Figure 2: PDDL example for preparing to open a garage door, which demonstrates the various complexities of nested non-deterministic effects.

As alluded to in the previous section, we can view the effect of an action as an *and-or* tree (demonstrated for our example in Figure 3). While this analogy is useful for some aspects, such as defining the realizations of an effect as $\mathcal{R}$, it is important to recognize the distinction from a Boolean formula represented as an *and-or* tree: **the effect of an action is not a Boolean function to be evaluated**. The biggest ramification of this is that we evaluate the effect in a top-down manner rather than bottom-up with the leaves. We elaborate on this point further in Section 3.3. Also, note that unlike a common *and-or* graph, we cannot simplify it by operations such as merging parent-child *or-or* node relations or collapsing seemingly tautological subtrees (such as the bottom *oneof* node in Figure 3), as each *oneof* node has a unique determiner associated with it.

3.1 State-Context Mapping

In a practical system, contextual information for the execution, above and beyond the planner's abstract view of the world, must be retained. The first key challenge that must be overcome is the correspondence between the planner's view (state) and the information used during deployment (context). The execution state is a set of fluents that hold and are important for the planner to decide which actions can be executed. The context C is an aligned assignment of values

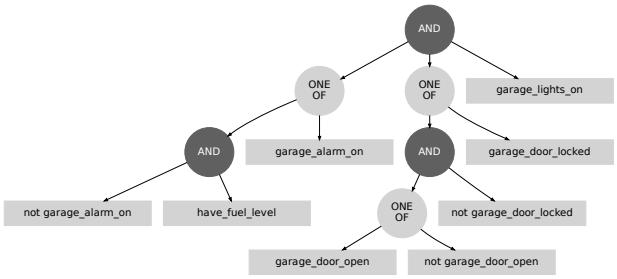


Figure 3: Effect tree of the action in Figure 2.

$C : \mathcal{F} \rightarrow Dom$, where Dom is an arbitrary (and possibly open ended) domain.

The alignment can be demonstrated with our running example: the state contains a fluent `(have_fuel_level)` $\in \mathcal{F}$ denoting the level of fuel in the car. From the planner's viewpoint, the only important result from preparing the car to exit the garage is whether the system knows the fuel level or not (regardless of its value). The fuel level itself may play a role in the execution and determination of other actions, and so the context value associated with the fuent (e.g., $C($`(have_fuel_level)`$) = 10L$) becomes important. For example, there may be another action, `check_fuel` that has a precondition `(have_fuel_level)` and non-deterministically results in either `(sufficient_fuel)` or `(not (sufficient_fuel))` (the former of which could act as a precondition for driving).

The advantage of splitting the execution information is twofold. First, it situates the planner at the right level of abstraction with the context maintained separately from the state. Second, it allows an interconnection between the plan and complex objects (like a web call result, agent coordinates, etc.) which would be impractical to represent in the planning domain. Maintaining the context alignment for all the reachable states is one of the challenges in the outcome determination, and described further in Section 3.3.

3.2 Action Execution -vs- Determination

The aim of contingent plan execution is running action callbacks associated with each state to achieve a specified goal state. However, in a non-deterministic world, an action can cause one of multiple effects on the state and context. Moreover, in the real world, the effects may not be fully observable and may be too complex to be modeled perfectly. Therefore, an essential part of the plan execution is deciding which of the effects best describes the real world change, i.e. the process of outcome determination.

This leads us to a natural decomposition of action execution into two phases (steps 3 and 4 from Figure 1 respectively). First, running the function callback that implements the real process in the outside world (e.g. calling a web service), which gives the system a function call result (e.g., a response code with a payload of information from a web service). Second, the outcome determination, which processes the function call result to update the execution state and context. Note that the outcome determination is a complex multistep process described in detail in the following section.At a high level, the determination process involves running the appropriate callback functions in the appropriate order to establish what realization should be used to progress the state of the world and solution status.

The separation between action execution and outcome determination is not just conceptually attractive from the standpoint of clearly separating distinct functionality in the implementation; it also provides a natural means of encouraging better declarative models. With the evaluation of an effect focused on computing what has changed (as opposed to making additional changes to the real world), complex and error-prone action models are avoided in lieu of multiple actions, each with a clearly defined purpose. The core anti-pattern that is avoided is the strategy of embedding aspects of an agent policy directly within an action effect (e.g., if a determiner selects one branch, then further actuation occurs that influences the environment).

In theory, the callback function and outcome determiners could be given the entire state and context. However, such a practice would result in an error-prone and hard to debug system. Therefore, *only the context subset claimed in an action's precondition is accessible for the action execution*: $C_a = \{C(f) | f \in \mathrm{Pre}_a\}$. This drastically reduces the potential for model mismatch due to modeling errors between the planning view of state, and full view of context. Providing the full context and state would be akin to using global variables exclusively in software development – a practice largely viewed to be error-prone and undesirable.

3.3 Complex Outcome Determination

The outcome determination is a fundamental part of action execution within a contingent plan. The role of determination is to decide on which realization $r \in \mathcal{R}(a)$ of the action a has occurred after the action callback AEF_a.

The action realization r is made up of ADD_r and DEL_r sets. During the outcome determination, those sets are recursively calculated from the action effect tree, and use the DET callbacks to compute these values for updating both the execution state and context.

Recursive Effect Processing The effect tree consists of nodes with two kinds of operators. First, the *and* operator $\bigwedge_\varphi$ with sub-formulae φ_i can be recursively calculated as:

$$ADD_{\bigwedge_\varphi} = \bigcup_{\varphi_i} ADD_{\varphi_i} \qquad DEL_{\bigwedge_\varphi} = \bigcup_{\varphi_i} DEL_{\varphi_i}$$

Second, the *or* operator $\bigoplus_\varphi$ and its selected child φ_j :

$$ADD_{\bigoplus_\varphi} = ADD_{\varphi_j} \qquad DEL_{\bigoplus_\varphi} = DEL_{\varphi_j}$$

The effect tree leaves φ_L directly define the ADD_{φ_L} and DEL_{φ_L} sets. With this recursion defined, the action realization update sets are calculated as update sets for the effect's top-level root node. The crucial part of outcome determination is the $\bigoplus_{\varphi_j}$ selection. In practical systems, the selection can be a time consuming service call. For instance, the agent can use an outcome determiner that performs a remote service call to a deployed recognition model for detecting entities in a scene; is there an object in the driveway or not,

is it dark enough outside to necessitate headlights, etc. Such calls can take a long time to execute (due to network latency and complexity of the computation), and therefore exhaustive calculation over the whole tree (which can be full of such expensive calls) might not be suitable.

Thanks to the tree structure, effects can be processed top-down, evaluating only the nodes that can contribute to the realization. For example, each *oneof* node needs only one sub-tree to be fully evaluated, and processing of that one sub-tree will only begin once the determiner for the *oneof* node has identified that it is the right one to proceed with. On the other hand, all of the sub-trees of an *and* node must be evaluated, which can be done in parallel (recall that the resulting update sets are not contradicting by definition). This process is depicted in the following algorithm:

Algorithm 1 Parallel Nested Determination algorithm

```
1: procedure ParallelNested(node)
2:     if node.type = leaf then
3:         ProcessLeaf(node)
4:     else if node.type = oneof then
5:         child = DET_node()                    ▷ Run determiner
6:         ParallelNested(child)
7:     else if node.type = and then
8:         apply_async(ParallelNested, node.children)
```

Note that on line 8, the children of an *and* node are concurrently processed, while on lines 5-6, the determiner for a *oneof* node is run until completion before recursing. Line 3 encapsulates the recursive ADD and DEL computation.

Retaining the full complexity of action effects thus gives us two key improvements on efficiency: (1) the ability to avoid evaluating sub-trees that correspond to outcomes a determiner deems did not occur; and (2) the ability to run determiners in parallel when they represent sibling sub-trees of an *and* node in the effect graph.

Dependencies Between Effects Some of the determiners may depend on finished execution of some other determiners. For instance, in our garage example, the status of the garage door being open or not is only relevant if the door is unlocked. In more extreme examples, a determiner may not be executable at all if its parent determiner did not resolve in a way that enables the child (i.e., having the right outcome selected). Such dependencies can prevent full parallel execution of the determiners simultaneously which would make our recursive tree execution necessary to get any parallelism and the associated speedup.

State and Context Updates With the realization update sets prepared, the new state after the action execution can be calculated as per usual: $S_{i+1} = (S_i \setminus DEL_r) \cup ADD_r$.

Context C is the other part of execution information that also has to be updated during the outcome determination. The new context values are produced by the determination process directly along the tree traversal described above (i.e., as part of the DET callback functions), and can additionally make use of the AEF action response. Our running example demonstrates one possible update: the fuel level will be set during the determination process given the information computed by $AEF_{\texttt{prepare\_garage\_car\_exit}}$. Formally, a realization $r \in \mathcal{R}(a)$ will have context updates C_r defined as:

$$\{C(f) = val \mid f \in ADD_r\} \cup \{C(f) = \bot \mid f \in DEL_r\}$$

The assignment of $C(f) = val$ is defined by the determiner callback for the effect tree's leaf nodes, DET_{φ_f} (not all fluents have context necessary for execution, in which case we set $C(f) = \bot$). Notice that such updates force the precise alignment of context and state, which is necessary for the system to function properly during execution.

Dependencies introduced by real world mechanics can cause difficulties for context value updates. For instance, fluent `(have_position)` and its context value $[x, y]$ may represent an agent position (and the validity of its reading). Then, fluents like `(have_x)` and `(have_y)` and their context values $[x]$ and $[y]$ correspond to separate position sub-components. An action changing a single sub-component for fluent `(have_x)` must also update context value of `(have_position)` which breaks action encapsulation and may lead to human errors during development.

Effective modeling of such real world dependencies is an interesting challenge that we keep for future work.

Completing Determination Once we have completed the determination process for action a, we are left with a new state s, context C, and realization $r \in \mathcal{R}(a)$. Given the current node n of the contingent plan, we compute the updated node as $n' = next_n(r)$. The final stage of the executor is to store all of the newly computed information (s, C, n') in the centralized database (step 5 in Figure 1).

4 Evaluation

To demonstrate the effectiveness of the proposed effect processing, we focus on the time it takes to do the determination. This speaks directly to the core novelty of our work: the execution of actions with complex nested effects.

We simulated the time to evaluate the outcome of three representative effect trees, which are depicted in Figures 4, 5, and 6. The size and the structure of the trees remained exactly the same during the simulation and only the outcomes and times needed to run the determiners changed.

For the purpose of this section, we will call our approach the *parallel nested* algorithm. Recall that it evaluates each *and* node in parallel and it uses the nested structure of the graph so it recursively evaluates only one sub-tree of an *oneof* node. We compared it with three naive algorithms:

1. *parallel flat*: does not care about the tree structure at all and evaluates all the determiners in parallel and then evaluates the outcome (which takes negligible time compared to the determiners)
2. *sequential nested*: the same as our proposed approach except the *and* nodes are not evaluated in parallel but sequentially (representing the effect a single core machine would have on computation)
3. *sequential flat*: the same as *parallel flat* but evaluates the determiners sequentially

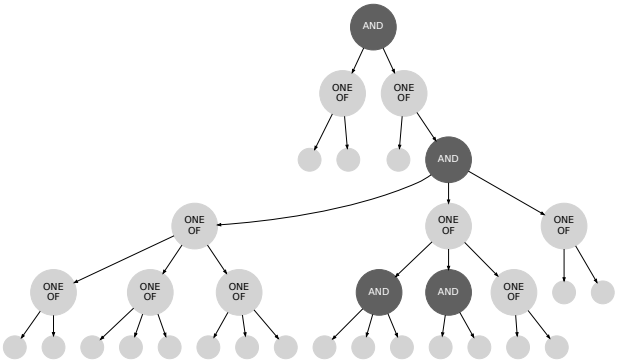


Figure 4: General complex effect tree, which is the most interesting from the practical application point of view. It could be, for example, a slightly more advanced version of the graph in Figure 3.

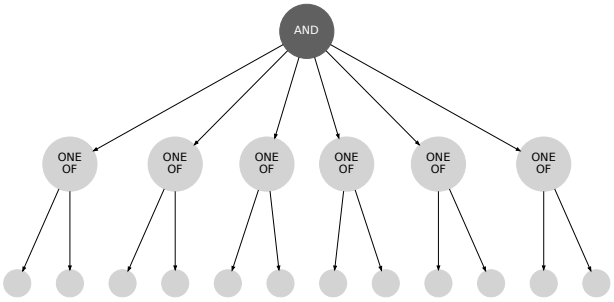


Figure 5: Flat effect tree that serves as a benchmark for how well the algorithms scale with growing number of determiners, that could be run in parallel.

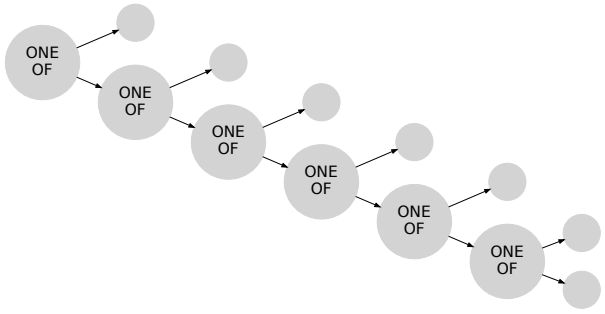


Figure 6: Deeply nested effect tree that serves as a benchmark for how well the algorithms scale with growing number of nested determiners.

Let us assume in this section, that all the determiners can run in an arbitrary order. If running some of the determiners required other determiners to run before, we could not use the flat algorithms. In this case, running a determiner in a node requires all of the determiners in its parent nodes to run prior, and the nested algorithms are the natural way to evaluate the tree.

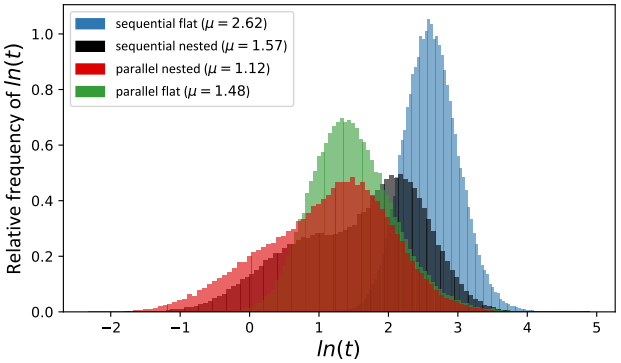


Figure 7: Distributions of logarithm of time to determine outcome of effect tree in Figure 4 (means in legend).

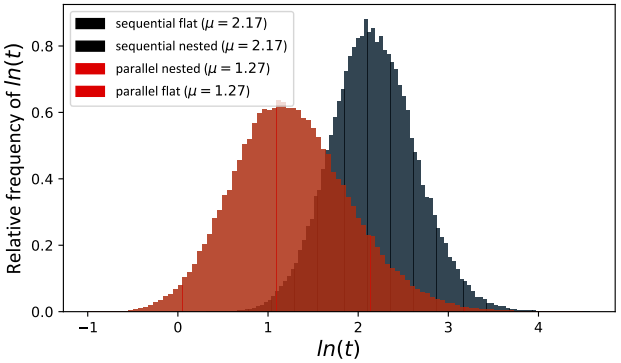


Figure 8: Distributions of logarithm of time to determine outcome of effect tree in Figure 5 (means in legend).

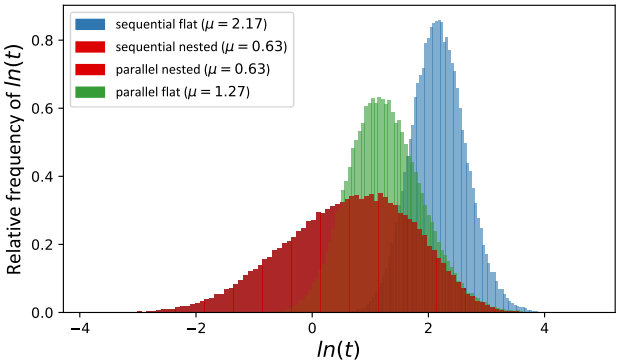


Figure 9: Distributions of logarithm of time to determine outcome of effect tree in Figure 6 (means in legend).

We restrict our attention only to the time it takes to evaluate the determiners in *oneof* nodes since this is the most expensive operation. We assume these times are all independent following *LogNormal(0,1)* distribution. This was chosen, as it is a reasonable distribution for the time to make an API call in our experience with the deployed system. To

gain a better perspective the actual running time of one determiner can be on the order of milliseconds to seconds. We also need to simulate the different outcomes. We do this recursively going top down in the tree selecting nodes that will belong to the outcome. The only decision we have to make in the process is what child of a previously selected *oneof* node to select next. We do this independently on selections in the previous *oneof* nodes so that all the child nodes have equal probability of being selected.

We run 100,000 simulations, sampling the determiner execution times and the outcome at each step. Histograms of the logarithm of simulated times for the different algorithms are depicted in Figures 7, 8, and 9. We chose to display histograms of $ln(t)$ instead of the actual times for clarity when comparing them with the histogram of log-times to run a single determiner, which would be depicted just as a probability density function of a standardized normal distribution.

Obviously we would expect the parallel versions to be faster than their sequential counterparts. The interesting comparison is between the *parallel nested* and *parallel flat* approaches. It is clear that the *parallel flat* will have to call on average more determiners, so it is more computationally expensive, but it is limited only by the time it takes to run the slowest determiner in the tree whereas the *parallel nested* algorithm waits for determiner in an *oneof* node to finish before running any determiner in its sub-trees. However, as we can see from the histograms, the *parallel nested* approach still outperforms *parallel flat* on average since it often does not have to call the slowest determiner at all. This is perhaps quite surprising considering all the determiner times were drawn from independent identical distributions.

In each step of the simulation, we used the same outcome and sample of determiner times for all the four algorithms, so it is not a surprise that the histograms of both parallel and both sequential methods coincide in Figure 8, since the *parallel flat* and *parallel nested* algorithms are the same for the graph in Figure 5 as are their sequential counterparts. This is due to the fact that there is no nested structure in the *oneof* nodes which could be exploited by the nested algorithms. The same thing happens for *sequential nested* and *parallel nested* histograms in Figure 9. This is because there are no *and* nodes in the graph, so there is no chance for the algorithms to run them in parallel.

Generally, we find that following the complex effect structure for determination, and using parallel execution whenever possible, represents a clear advantage.

5 General Execution Monitoring

In this paper, we have assumed that the executor adopts a view of accurate determination and proceeds without a notion of active monitoring on any discrepancies that may exist between the expected and observed states. However, our work in no way prohibits the use of a high-level EM that is capable of detecting full discrepancies with the anticipated state, and re-adjusting (or re-planning) as necessary. There is a rich array of work on EM systems (some of which is discussed in Section 6) that focus on only relevant aspects of the state in order to ensure the plan remains valid, and we consider this line of work to be fully complementary to ours.

The most natural place for integration would be in step 5 of the architecture presented in this paper. Rather than simply updating the database with the new state, context, and node in the solution graph, we instead defer to an EM system that is capable of detecting relevant discrepancies and then adjusting as necessary: either by finding the most appropriate point in the plan from which to continue execution or replanning entirely. Note, however, that we have inherently adopted a notion of contingency and modeled uncertainty, as our plans are fully contingent and not sequential. Thus, discrepancies will only arise if we have indicators aside from the determiners that are capable of computing the value of context variables and surfacing any potential discrepancies.

6 Related Work

Much execution monitoring work focuses on the theoretical properties of the system without diving into the details of how the physical system should map to the planner's abstraction of the environment, or forgo entirely the notion of a framework for determining action outcomes. TPOPEXEC (Muise, Beck, and McIlraith 2013) addresses the theoretical properties of deterministic temporal partial order plans (as opposed to the conditional plans we consider). The Kirk and Drake systems (Block, Wehowsky, and Williams 2006; Conrad and Williams 2011) similarly focus on partial order temporal plans, but with choice points in the execution (akin to contingent solutions). A similar approach is employed by the Razor system for compiling contingent plans in an information gathering setting (Friedman and Weld 1997). However, these works mainly deal with compiling one form of plan with unrealized choice points, to an executable or *dispatchable* form: the complexity of non-determinism is trivial (compared to the full nesting of *and* and *oneof* that we consider in Section 3.3), and there is no focus on the mapping to realized systems (i.e., correspondence between context and state, and separation of action execution and realization). Other work extends the notion of uncertainty to temporal durations (Karpas et al. 2015), but again this is mainly theoretical in nature and focuses on a separate set of challenges.

Closer to our work, the IXTET-EXEC system (Lemai and Ingrand 2004) focuses on some of the challenges an executor faces when plans are deployed; the key difference being that their focus is on temporal actions and not contingent plans with modeled uncertainty (i.e., complementary aspects of execution). The system is built on the OpenPRS procedural executor (Ingrand 2015), and unexpected failures to the execution are handled through replanning and plan repair approaches. Languages for executors have also been proposed, such as PLEXIL (Verma et al. 2005) and RMPL (Williams et al. 2003). Similar to the work cited above, the focus of these languages is to place temporal-based plans in a dispatchable form, with the focus on adhering to the semantics at the planning level of abstraction and temporal consistency of the execution.

The languages of PLEXIL and Esterel (Berry and Cosserat 1984) are programming languages for autonomous systems. Their relation to our work is in the output format of the contingent plans we produce, but not in the higher-level philosophy of declarative modeling. Of the two languages,

PLEXIL uses a representation closer to the complex nested effects we describe (i.e., a graph of nodes with key similar interpretation). Both PLEXIL and Esterel are well-defined programming languages with rich expressibility. Our work aims at addressing a different set of challenges with respect to execution: namely the specification and handling of complex action effects, and the challenges / opportunities surrounding the connection between state and context.

In terms of the state and context mapping that we propose, whenever the context C is defined for a fluent f (i.e., $C(f) \neq \perp$), then we can view f as representing the fact that we know the value of a particular variable with a rich domain. This mirrors the idea of the $Knows$ predicate in (Scherl and Levesque 1993) and $KnowIf$ (KIF) variables in (Brenner and Nebel 2009). Also related is the recent work on integrating PDDL plan execution with the CLIPS rule-based production system (Niemueller, Hofmann, and Lakemeyer 2018). In this work, various models are defined, including the planner model (which corresponds to our planning state view) and world model (which corresponds loosely to our context view). Key differences, however, include our focus on non-deterministic settings and the direct relation to implementations of action execution and outcome determination.

The connection between high-level action specifications and low-level sensor / behaviour modalities is described in the work focusing on Object-Action Complexes (Krüger et al. 2011). The connection to our work is in the correspondence between planning actions and the action execution functions we use to realize them. However, there is little more to be drawn from the parallel, aside from the proposed approach of maintaining two views on the world.

Similar approaches can be seen in the robotics community through works such as the KNOWROB, SkiROS, and ROSPlan systems (Tenorth and Beetz 2009; Rovida et al. 2017; Cashmore et al. 2015). The robotics focused systems similarly solve the task of linking the planning state and the execution context. Differences include their focus on temporal execution and the monolithic view of action execution / determination. Perhaps the most mature of the existing work, ROSPlan, accepts a variety of input plan specification languages, including simple sequential plans, the Esterel plan language (which represents a temporal plan without the uncertain contingencies we aim to support), contingent plans represented as state-to-action policies, and Petri Net Plans (Ziparo et al. 2008). Both the contingent plan and Petri Net Plan representations fail to capture much of the sophistication we introduce in this paper, including the separation of action execution / determination, and the nested functionality of non-deterministic effects. Conversely, aspects such as temporal action execution, loops, and interrupts are components that the various ROSPlan interfaces are capable of expressing that we forgo in our present work.

Similar to the discussion in Section 5 on embedding our work into a larger EM framework, there is potential for us to extend a system such as ROSPlan to capture our methodology for the robotic setting. The advantage would largely come in the form of the improved expressivity in action effects for the declarative specification of robot behaviour.

7 Summary

We have presented a high-level architecture for executing contingent plans containing actions with complex nested effects to model uncertainty, and empirically demonstrated the improvements that our method can bring. In support of defining an effective execution agent, we also detailed a crisp connection between the abstraction used by a planner and the real-world view, and introduced a method for effective modeling by distinguishing the process by which an agent affects the world from the process which updates its understanding as a result. Our work fills a critical gap in the pipeline of using automated planning in practical settings. The principles we put forward in this work stem from the lessons learned in developing and deploying virtual agents driven by contingent plans on prototype applications in an industrial setting. In particular, the need for complex action effects, and the effective determination of them, was critical to the successful execution of the plans. We plan to open source a generalized version of our execution framework along with the publication of this work. We conclude by detailing some of our plans for future work in the area.

7.1 Future Work

There are many directions that we would like to explore, now that we have introduced a foundation for the effective execution of contingent plans. Here, we list two of the most compelling avenues for future work.

Imperfect Models Following on the connection to more general execution monitors, there is an interesting disconnect between modeled uncertainty, and observed discrepancies. If monitors detect that the execution results in a determination that is incorrect relative to the true context values, then *either the modeled abstraction of uncertainty or the determination process must be erroneous*. Rather than simply following the standard EM practice of continuing the execution of another part of the plan (or replanning in the worst case), we have the opportunity to repair the model directly. This would include one of two options: (1) changing the model directly to account for new possible uncertainties in the action effect (and thus requiring new determiners to be introduced); or (2) improving the determination process directly so that incorrect outcome predictions are improved over time. We have begun to explore the latter possibility and have seen promising results for certain types of determiners.

Advanced State-Context Mapping A major area of future work is the connection between state and context, where some of the key challenges are identified in (Frank 2015). Currently, we only consider a subset of the fluents to have expressive domains associated with them. However, in general execution, fluents in the planning model could represent a complex function of context (e.g., an inequality over real-valued variables). Conversely, a single context variable could yield multiple fluents (e.g., `(low_fuel_level)` and `(high_fuel_level)` for our running example). The mapping between context and state becomes substantially more difficult in this setting, but addressing it is an important step towards creating a more powerful executive.

References

Berry, G., and Cosserat, L. 1984. The ESTEREL synchronous programming language and its mathematical semantics. In *Seminar on Concurrency, Carnegie-Mellon University, Pittsburg, PA, USA, July 9-11, 1984*, 389–448.

Block, S. A.; Wehowsky, A. F.; and Williams, B. C. 2006. Robust execution on contingent, temporally flexible plans. In *AAAI*, volume 2006, 802–808.

Bonet, B., and Geffner, H. 2014. Belief tracking for planning with sensing: Width, complexity and approximations. *J. Artif. Intell. Res.* 50:923–970.

Brenner, M., and Nebel, B. 2009. Continual planning and acting in dynamic multiagent environments. *Autonomous Agents and Multi-Agent Systems* 19(3):297–331.

Camacho, A.; Muise, C.; and McIlraith, S. A. 2016. From fond to robust probabilistic planning: Computing compact policies that bypass avoidable deadends. In *26th International Conference on Automated Planning and Scheduling*.

Cashmore, M.; Fox, M.; Long, D.; Magazzeni, D.; Ridder, B.; Carrera, A.; Palomeras, N.; Hurtós, N.; and Carreras, M. 2015. Rosplan: Planning in the robot operating system. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2015, Jerusalem, Israel, June 7-11, 2015.*, 333–341.

Conrad, P. R., and Williams, B. C. 2011. Drake: An efficient executive for temporal plans with choice. *J. Artif. Intell. Res.* 42:607–659.

Frank, J. 2015. Reflecting on planning models: A challenge for self-modeling systems. In *Autonomic Computing (ICAC), 2015 IEEE International Conference on*, 255–260. IEEE.

Friedman, M., and Weld, D. S. 1997. Efficiently executing information-gathering plans. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI*, 785–791.

Geffner, H., and Bonet, B. 2013. *A Concise Introduction to Models and Methods for Automated Planning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Geffner, T., and Geffner, H. 2018. Compact policies for fully observable non-deterministic planning as SAT. In *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018.*, 88–96.

Ingrand, F. 2015. Open procedural reasoning systems (openprs). `https://git.openrobots.org/projects/openprs`. Accessed: 2018-07-23.

Karpas, E.; Levine, S. J.; Yu, P.; and Williams, B. C. 2015. Robust execution of plans for human-robot teams. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling, ICAPS 2015, Jerusalem, Israel, June 7-11, 2015.*, 342–346.

Krüger, N.; Geib, C. W.; Piater, J. H.; Petrick, R. P. A.; Steedman, M.; Wörgötter, F.; Ude, A.; Asfour, T.; Kraft, D.; Omrcen, D.; Agostini, A.; and Dillmann, R. 2011. Object-action complexes: Grounded abstractions of sensory-motor processes. *Robotics and Autonomous Systems* 59(10):740–757.

Lemai, S., and Ingrand, F. 2004. Interleaving temporal planning and execution in robotics domains. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, 617–622.

Muise, C. J.; Beck, J. C.; and McIlraith, S. A. 2013. Flexible execution of partial order plans with temporal constraints. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 2328–2335.

Muise, C.; Belle, V.; and McIlraith, S. A. 2014. Computing contingent plans via fully observable non-deterministic planning. In *The 28th AAAI Conference on Artificial Intelligence*.

Muise, C.; McIlraith, S. A.; and Beck, J. C. 2012. Improved Non-deterministic Planning by Exploiting State Relevance. In *The 22nd International Conference on Automated Planning and Scheduling*, The 22nd International Conference on Automated Planning and Scheduling.

Niemueller, T.; Hofmann, T.; and Lakemeyer, G. 2018. Clips-based execution for pddl planners. In *Workshop on Integrated Planning, Acting, and Execution (IntEx'18)*.

Palacios, H., and Geffner, H. 2009. Compiling uncertainty away in conformant planning problems with bounded width. *J. Artif. Intell. Res.* 35:623–675.

Rovida, F.; Crosby, M.; Holz, D.; Polydoros, A. S.; Großmann, B.; Petrick, R. P.; and Krüger, V. 2017. Skirosa skill-based robot control platform on top of ros. In *Robot Operating System (ROS)*. Springer. 121–160.

Scherl, R. B., and Levesque, H. J. 1993. The frame problem and knowledge-producing actions. In *Proceedings of the 11th National Conference on Artificial Intelligence. Washington, DC, USA, July 11-15, 1993.*, 689–695.

Tenorth, M., and Beetz, M. 2009. KNOWROB - knowledge processing for autonomous personal robots. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems, October 11-15, 2009, St. Louis, MO, USA*, 4261–4266.

Verma, V.; Estlin, T.; Jónsson, A.; Pasareanu, C.; Simmons, R.; and Tso, K. 2005. Plan execution interchange language (plexil) for executable plans and command sequences. In *International symposium on artificial intelligence, robotics and automation in space (iSAIRAS)*.

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):212–237.

Ziparo, V. A.; Iocchi, L.; Nardi, D.; Palamara, P. F.; and Costelha, H. 2008. Petri net plans: a formal model for representation and execution of multi-robot plans. In *7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008), Estoril, Portugal, May 12-16, 2008, Volume 1*, 79–86.

A Hybrid Planning and Execution Approach Through HTN and MCTS

Xenija Neufeld
Faculty of Computer Science
Otto von Guericke University
Magdeburg, Germany,
Crytek GmbH, Frankfurt, Germany

Sanaz Mostaghim
Faculty of Computer Science
Otto von Guericke University
Magdeburg, Germany

Diego Perez-Liebana
School of Electronic Engineering
and Computer Science
Queen Mary University of London
London, United Kingdom

Abstract

Many planning environments require from an agent to show a combination of long-term strategical behavior and reactive short-term tactical behavior. In order to combine planning on both hierarchy levels and to detect potential failures, they also require an interleaved planning and execution approach. In this work, we propose a hybrid planning approach with a Hierarchical Task Network planner being responsible for strategical planning and Monte Carlo Tree Search taking over the tactical decision-making. We describe a possible way to connect these layers and a monitoring system that is able to detect failures on higher hierarchy levels during execution. The proposed approach is tested in a Real Time Strategy game that offers a highly-dynamic and non-deterministic multi-unit environment.

1 Introduction

Many real-time planning problems such as rescue missions or collaboration of multiple industrial robots require planning approaches that combine strategical long-term planning and micro-action control and execution. Real Time Strategy (RTS) games provide complex simulation and test environments for such approaches since they are challenging in many ways. Not only does an intelligent agent playing such games need to compute a plan in a very short time (usually within milliseconds), it also needs to operate in a huge search space, plan for multiple heterogeneous units that execute durative actions and act in a non-deterministic environment. The non-determinism comes from the environment itself and the actions of the agent and its opponent. Additionally, most RTS games provide only partial observability of the environment.

In order to perform well in RTS games, an agent needs to incorporate both hierarchy levels: strategical planning (for example how many buildings and mobile units to build and when to start an assault) and tactical micro-management on a unit level (for example how to position units). Additionally, the agent needs to constantly monitor the environment and stay reactive to its changes and the opponent's actions (for example instead of continuing to build new buildings, defend its own base if the opponent attacks first).

Monte Carlo Tree Search (MCTS) approaches have shown to perform well in big search spaces of several games (Ishihara et al. 2016; Sironi et al. 2018; Perez, Rohlfshagen, and Lucas 2012). However, due to a huge branching factor in RTS games, MCTS usually can only plan a few steps ahead. Thus it can be used for one hierarchy level only (usually the tactical level). On the other hand, Hierarchical Task Networks (HTN) provide a way to create high-level strategies decreasing the search space early in the search process but are not suitable for micro-management due to large branching factors and the high cost of manual authorship of the planning domain. Furthermore, because of very high dynamics of an RTS game environment, *long-term* HTN plans are very likely to fail during execution whereas MCTS returns a single action to be executed in the *current step*.

In order to profit from the advantages of both approaches we propose a hybrid approach that allows for strategical and tactical planning, monitoring, and execution. We use an HTN planner on a high level, MCTS on the unit-level and a monitoring system that detects plan failures at execution time and triggers re-planning. We test the proposed approach in microRTS[1] (Ontañón 2017b; Ontañón et al. 2018), a simplified research RTS environment which takes care of the execution of the actions provided by MCTS. Although microRTS provides a partially-observable mode, dealing with partial observability is out of scope of this paper.

The hybrid planning happens in the following way: the HTN planner creates a high-level plan, such as *collect resources, build barracks, create military units, attack the opponent*. Each of these tasks is defined by a function describing the objectives that are optimized by this task. Afterwards, this plan is passed to the agent for execution and the current task is forwarded to MCTS to search for an optimal move. Since MCTS operates with evaluation functions, it selects the function of the current HTN task to evaluate the game states after its rollouts (see Section 2.2). At the same time, the agent monitors the game environment, re-evaluates the HTN tasks and triggers re-planning, if needed.

In the approach proposed here, MCTS switches between different evaluation functions as indicated by the planner. Our expectation is to see a more complex behavior than us-

[1] microRTS: https://sites.google.com/site/microRTSaicompetition

ing a singe evaluation function. The main contributions of this work are: a proposal of a way to combine HTN planning with MCTS, a planning and execution architecture using this hybrid planning technique and the comparison of this approach against a pure MCTS agent that uses a single evaluation function throughout the whole game, a pure HTN agent and an agent that combines strategical and tactical behavior selection.

The rest of the paper is structured as follows: Section 2 gives some insights into the test environment and the approaches used for our work. Section 3 shows related approaches followed by Section 4 that describes the hybrid planning architecture and the interleaved planning, monitoring, and execution processes. Section 5 gives insights into the experimental work and its results. Finally, Section 6 concludes our findings and outlines future work.

2 Background

2.1 HTN Planner

Similarly to (Höller et al. 2018a; 2018b), we define a Hierarchical Task Network (HTN) planning problem by the tuple $p = (F, C, A, M, s_I, c_I)$. F is a set of state variables or *facts*. Starting from an initial state $s_I \in 2^F$, the planner creates a plan by decomposing the initial *compound task* $c_I \in C$ into further compound or *primitive tasks* (A). Primitive tasks usually represent actions that an agent can execute and thus cannot be further decomposed. Compound and primitive tasks build a *task network* which is defined as $tn = (T, \prec, \alpha)$ with T being a set of possibly empty identifiers and $\alpha : T \rightarrow A \cup C$. When decomposing a compound task $c \in C$, the planner selects one of the decomposition *methods* ($m \in M$) that can decompose the given task. Methods are defined as a triple (c, pre, tn) with the compound task c that is decomposed, a network of *subtasks* tn resulting from the decomposition and *preconditions* $pre \in 2^F$ under which the method can be applied. Similarly, we define primitive tasks by $(pre, add, del, post, ef)$ with preconditions, *effects* that add or delete facts from a state ($add \in 2^F$ and $del \in 2^F$ respectively) after executing a task.

Additionally to the definitions provided in (Höller et al. 2018a; 2018b), we add preconditions to method definitions, and postconditions and evaluation functions ef to primitive tasks. Postconditions $post \in 2^F$ define what facts are required to hold in the state in which the task finishes. We use postconditions during the execution phase to detect the end of a durative task (see Section 4.2). Evaluation functions of primitive tasks are used as an interface between HTN and MCTS and are described in more detail in Section 4.1. The HTN planner implemented in this work is a total-order planner similar to one of the most known HTN planners SHOP (Nau et al. 1999) and SHOP2 (Nau et al. 2003).

2.2 MCTS

Monte Carlo Tree Search (MCTS) algorithms search for optimal solutions of Multi-armed Bandit (MAB) problems balancing between exploration and exploitation (Browne et al. 2012). They combine tree search and Monte Carlo simulations. Searching through the space of (game) states, the tree starts from a root node, *selects* a leaf node following a *tree policy* and *expands* it executing the corresponding action. The tree policy consists of selecting and expanding nodes and is usually performed until reaching a terminal criterion. Each node s stores the number of times that it was visited $N(s)$, the number of times that a certain action a was applied in this node $N(s, a)$, and the average reward $Q(s, a)$ gained from its visit.

After expanding the tree, a simulation runs from one of the leaf nodes following a *default policy* (for example random selection) until a certain simulation depth or another termination criterion is reached. The leaf node is then evaluated by an *evaluation function* and the result is backpropagated updating the statistics of the nodes that led to this node. Finally, the action a^* leading to the *optimal* child of the root node is selected. Therefore, most MCTS algorithms balance between exploration of nodes that were visited less often and exploitation of the nodes that led to high results of the evaluation function. A common approach to achieve this balance is Upper Confidence Bound (USB1) as a tree policy (Auer, Cesa-Bianchi, and Fischer 2002) that is shown in equation 1. By increasing K, a higher weight can be given to the second part of the equation preferring exploration over exploitation. A detailed survey on further MCTS methods is given in (Browne et al. 2012).

$$a^* = arg\ max_{a \in A(s)} \left\{ Q(s,a) + K \sqrt{\frac{ln\ N(s)}{N(s,a)}} \right\} \quad (1)$$

When using MCTS for an RTS game, the algorithm needs to find an optimal combination of actions of multiple units and thus it is dealing with a Combinatorial MAB (CMAB) problem. A detailed description of CMAB sampling techniques for MCTS is given in (Ontanón 2017a). One of the most effective techniques for microRTS specifically has shown to be naiveMCTS (Ontañón 2013). This sampling method (naively) assumes that the reward distribution $R(x_1, ..., x_n)$ of multiple variables x_i can be approximated as $\sum_{i=1}^{i=n} R_i(x_i)$ – the sum of reward functions that depend on one variable each, breaking the CMAB problem into a set of MAB problems.

In (Ontañón 2013), naiveMCTS samples over combinations of unit actions. First, it uses an ϵ-greedy policy π_0 to select between exploration and exploitation (ϵ is the probability to select exploration and $1 - \epsilon$ for exploitation). Then, depending on the choice, it uses either an ϵ-greedy policy π_l for exploration selecting an action for each unit or a pure greedy policy π_g for exploitation selecting an action *combination*. This work has shown that naiveMCTS outperforms algorithms such as Upper Confidence Tree. We use this implementation in our work without any changes to it.

2.3 MircoRTS

microRTS is a research environment that represents a typical RTS game. Since 2017 it is used for the microRTS AI competition (Ontañón 2017b). The environment offers the possibility for AI agents to act as players providing them with

information on the game state. Each agent must return the actions that need to be executed by the player's units every frame, which lasts 100 milliseconds. If an agent exceeds this time while computing its decision, it is disqualified from the match. microRTS offers a game mode with partial observability limiting the information known to an agent simulating a fog of war. Furthermore, it provides a forward model enabling the use of Monte Carlo algorithms. In comparison to commercial games, it is simplified in the sense that it offers very basic visual representation of the game world. Additionally, for fast experimental tournaments, it can run without any visuals at all. Nevertheless, it offers the typical gameplay possibilities and requirements including resource gathering, unit creation, and assault resulting in a complex environment.

microRTS uses symmetric maps of different sizes ranging from an 8×8 cells map to a 256×256 cells map. Each player starts with a fixed number of units (usually a *base* and potentially some *workers*). Additionally, he may build *barracks* which then may produce the following military units: *light, heavy* and *ranged*. Workers as well as military units may attack opponent units within a unit-type range but only workers may gather *resources* that are initially placed on the map. Gathered resources must be brought to a base in order to create further units or buildings. Buildings as well as moving units have a certain amount of *hit points* which are decreased by an opponent's attacks.

An ideal game match would proceed in the following way: we either have enough resources or we have the possibility to collect enough resources in order to start building necessary buildings (in the case of microRTS – barracks). Once we have the required buildings, we can start creating military units and, finally, we can move our military units towards the enemy base and destroy his units.

Thus, on a high strategical level we need an opening (collecting phase), a middle-game (building phase) and an end-game (attack). Depending on factors like the size of the map or the reachability of the enemy base we need to refine these phases in different ways. For example, on a small map, our opponent will be very close and attack quickly. Thus, in the opening, we should create many workers and then switch to the end game. Or if the path to the enemy's base is blocked by resources, we need to create a way to walk through (by collecting the resources) before switching to the end game.

3 Related Work

Monte Carlo algorithms dealing with different hierarchy levels of planning in RTS games have been explored in multiple works. (Chung, Buro, and Schaeffer 2005) introduced *MC-Plan* using MCTS on the strategical level in the free software OpenRTS. Dealing with an abstract state space, it showed promising initial results in the area of RTS.

(Balla and Fern 2009) used the Upper Confidence Tree (UCT) algorithm in the game Wargus. This work focused on the tactical level of the game. Also using abstract states, the algorithm reasoned about groups of units where the management of individual agents was left to the game engine.

Another UCT approach was introduced by (Soemers 2014) in the more complex RTS game StarCraft. Here, UCT was used for the tactical battle layer having multiple additional systems taking care of the strategy, economy development and unit actions.

Other works that are based on UCT are (Churchill and Buro 2013) and (Justesen et al. 2014). The former implemented a UCT Considering Duration (UCTCD) algorithm which was used for tactical movement in StarCraft. Although this algorithm outperformed the built-in scripted AI of StarCraft, it did not perform well for big numbers of units. Two improvements of this approach were described in (Justesen et al. 2014). Here, the search space was abstracted first, by dealing with more complex scripts instead of micro-actions, and second, by assigning those scripts to clusters of units instead of dealing with single units. This approach outperformed the original UCTCD when controlling high numbers of units.

Further exploration of Monte Carlo algorithms in StarCraft has been done by (Uriarte and Ontañón 2014). This work focused on army maneuvering through MCTS. Therefore, it used abstract game states, a high-level forward model and abstract evaluation functions. This approach could not outperform the scripted built-in AI because it was not always able to search deep enough to find a winning plan. However, this work showed that the abstraction of the search space could reduce the branching factor while still providing meaningful actions.

The work by (Ontañón 2013; Ontanón 2017a) investigated multiple Monte Carlo techniques in microRTS focusing on the micro-action level. The results have shown that especially for problems with large branching factors the sampling strategy naiveMCTS outperforms other algorithms such as Alpha-Beta search or UCT. For this reason, our work uses naiveMCTS. Further improvements to naiveMCTS were made in (Ontanón 2016). Here, the tree search was guided by additionally taking into consideration a pre-learned probability distribution of unit-actions. The improved version outperformed the original naiveMCTS.

Another work that used an MCTS agent is described in (Sironi et al. 2018). Here, the agent was used to play different games of the General Video Game Playing environment. It is related to our work in that sense that (instead of using different evaluation functions for MCTS) they tuned different MCTS parameters, such as the exploration factor or the search depth, to improve an agent's performance in the games. Compared to the baseline agent, the adapted agents performed similarly or better in different games.

Different hierarchical approaches have been implemented for various RTS game environments.(Stanescu, Barriga, and Buro 2014) proposed a combination of hierarchical adversarial search on higher levels and the usage of either Alpha-Beta or Portfolio search for the generation and execution of micro-actions. In SparCraft – a StarCraft combat simulator - this approach outperformed Alpha-Beta search, UCT and Portfolio Search when dealing with a high number of units (more than 72).

Adversarial planning with HTNs (AHTN) was introduced in (Ontañón and Buro 2015). This work combined elements of game tree search and HTN planning dropping the assumption of a turn-based game and allowing durative and simul-

taneous actions. Tested in microRTS, it compared AHTNs of different hierarchy depths against each other and in-built agents (3 of them were based on Monte Carlo techniques). All AHTN agents with a depth higher than 1 (not only micro-actions) outperformed the in-built agents.

A different approach to planning in RTS with abstracted actions was introduced as Puppet Search (Barriga, Stanescu, and Buro 2015). Here, complex scripts forwarded the game state further than single actions and exposed choice points to the search algorithm. The scripts could use a local search or another optimization technique. First tested in StarCraft, Puppet Search performed similar to the best benchmark agent. Later, this work was extended with a convolutional neural network for script selection (Barriga, Stanescu, and Buro 2017) leaving only the low-level tactics to game tree search. This approach won the 2017 edition of the microRTS competition.

Works related to our monitoring part are for example (Weber, Mateas, and Jhala 2010) and (Gonzlez Dorado et al. 2018). In (Weber, Mateas, and Jhala 2010), a planning and monitoring approach for StarCraft (Weber, Mateas, and Jhala 2010) is described. It was using a Goal oriented Action Planner (GOAP) (Orkin 2006). The planner was creating actions and *expectations* of the world state after an action (similar to the post-conditions described in Section 2.1). After executing an action of a previously created plan, a discrepancy detector compared the expected game state against the actual game state and, if needed, triggered re-planning. This part is very similar to the monitoring and re-planning approach used in our approach. The work by Gonzlez Dorado et al. describes a three-layer architecture for cobots (robots that collaborate with humans). It recognizes opportunities and plan failures on the high level and reschedules tasks at runtime.

4 Hybrid Approach

Due to the fact that we need to operate with high-level strategies that can be decomposed into different lower-level tactics, HTNs seem to be a suitable approach to plan with in microRTS. However, the ideal scenario described in Section 2.3 is not very likely to happen if we are playing against a good enemy who either prevents us from building our units or is well-prepared for our attack. Thus, we need to stay reactive to changes in the environment and adjust our plan accordingly. For that reason, we propose interleaving planning and execution as described in the following sections.

4.1 Hybrid Planning

Additionally to the high dynamics of the environment, the large search space adds up to the complexity of the planning approach required for an RTS game. The search space grows with the map size, the number of units that a planner needs to plan for, and the number of actions that each unit can possibly execute. In order to decrease the search space, many previous works have acknowledged the need to use either *abstract* planner states or abstract actions (Barriga, Stanescu, and Buro 2015; Moraes et al. 2018; Churchill and Buro 2013; Justesen et al. 2014).

In a similar way, we propose using abstract states, preconditions, and effects in our HTN planner. Each abstract planner state contains information such as the current number of friendly units and resources, the number and locations of enemy buildings but no locations of mobile units since these might change very fast.

In the beginning of a match, the planner estimates the minimum desired numbers of units of each type for the given map size. Afterwards, following a data-driven approach, these numbers are used in combination with abstract planner states for precondition checks and effect propagation. For example the task *BuildBarracks* requires the agent to have *enough* resources to build the desired number of barracks. The exact number of resources can be computed at plan-time knowing the desired number of barracks and the cost for building barracks. Thus, the abstract planner state in which the preconditions for this task are checked, should have at least this number of resources (and thus the preceding task *CollectResources* should add this fact to the planner state.) As an effect of the building task, the planner can set the actual number of barracks in the following planner state to the desired one. Similarly, instead of pre-planning the exact position of the new barracks, we can save the abstract fact that they will be *reachable from* or *close to* a base (defining what distance is considered as *close* in regards to the map size).

At this point, we do not reason about the enemy's actions when planning on a strategical level. Due to the decreased size of the search space, the planning is fast enough to be performed within the time limits (100 milliseconds) set by microRTS. Instead, we rely on recognizing plan failures at run-time and re-planning, if necessary, as described in the next section. For future work, however, we might consider adding a high-level reasoning and counter-planning mechanism such as for example (Pozanco et al. 2018; Sailer, Buro, and Lanctot 2007) or (Ontañón and Buro 2015).

Although abstract states might be sufficient for strategical high-level planning, when it comes to tactical planning on a unit level, we need to consider the actual game state. However, performing both levels of planning with an HTN would first, require a lot of engineering effort when creating the low-level HTN domain for all possible combinations of unit actions and second, creating a plan of micro-actions for each unit would increase the planning time immensely. For that reason, we propose handing over the micro-action part to MCTS. That way, the HTN planner can create an abstract plan up to a certain hierarchy depth, forward the abstract plan steps to MCTS which can find the optimal actions for all units taking into account the currently executed HTN task.

In order to combine the high-level tasks of HTN with MCTS planning, we need to represent the tasks in a way that could be used by MCTS when searching for an optimal action distribution between units. For that reason, we propose representing each primitive task of the HTN by an evaluation function f_t that describes mathematically the effects of the task t. For example when executing the *BuildBarracks* task, the agent should try to maximize the number of his barracks. However, in such an environment, the agent would usually

need to optimize not only one but multiple objectives. For example, he would also want to minimize the distance between each of his mobile units and his buildings (defending them), to maximize the number of resources (continuing to collect them), and to maximize the hit points of all his units while minimizing the hit points of the opponent's units. For that reason, we propose defining the evaluation function f_t that represents a primitive task of an HTN using the weighted sum approach weighting the n evaluation functions that aim to optimize n different (possibly conflicting) objectives x_i as shown in equation 2.

$$f_t = \sum_{i=1}^{n} w_i f_i(x_i) \tag{2}$$

For this work, we have set the weights manually according to our own knowledge of the game and after performing some prior experiments. For future work, however, we consider learning the weights automatically and optimizing the effectiveness of the evaluation functions.

4.2 Interleaved Monitoring, Execution, and Re-planning

The interleaved planning and execution approach of our agent is shown in algorithm 1. In order to execute an agent's action in the game environment, microRTS calls the *GetPlayerAction* method at every time step – a so called frame. A *PlayerAction* contains the combination of actions of all units that can execute any action in this frame. Since unit actions are durative, a unit can start a new action only when the previous one is finished (for example when *building* was completed). The execution of a single unit action – which in this case means rendering the correct animation and counting the remaining frames until action end – is performed by microRTS itself. Our algorithm is responsible for triggering new unit actions when possible.

The three major parts of the algorithm are 1) updating the HTN planner's view of the world state in line 6, 2) creating the high-level HTN plan in line 25 and 3) assigning and triggering unit actions according to the current task through MCTS in line 38. Since the HTN planner uses *abstract* states and does not take into consideration actual locations of units, it is not necessary to update its world state in every frame. Instead, as shown in lines 5 – 8, aiming to optimize the computation time, we propose updating the world state only at a certain frequency (in our case, every 10 frames) or whenever the agent cannot start a new unit action and thus neither HTN nor MCTS do require any time for computation. In those frames where the algorithm needs to run the HTN planner as well as MCTS, we allow the HTN planner to use up to 80 out of 100 milliseconds leaving the remaining time for MCTS. This division of the budget comes from the fact that the HTN planner runs only if there occur changes in the game world that trigger a re-planning on a high level. Therefore, we can afford not giving much time to MCTS in this frame assuming that it will balance out its decision in the next frames having more computation time.

In the beginning of a game, the initial plan needs to be created. There exists no current task in line 13 yet and thus

Algorithm 1 GetPlayerAction()

```
1:  s ← current game state
2:  π ← current plan
3:  t ← current task
4:  f ← current evaluation function
5:  if can'tExecuteAnyUnitAction or isTimeToUpdate then
6:     UpdatePlannerWorldState()
7:     if cantExecuteAnyUnitAction then
8:        return noAction
9:     end if
10: end if
11: decisionMade ← false
12: replan ← false
13: if t ≠ nil then
14:    if t finished then
15:       {continue in line 20}
16:    else if t running and t valid then
17:       decisionMade ← true {proceed with MCTS}
18:    else {t invalid}
19:       replan ← true
20:    end if
21: end if
22: while decisionMade = false and belowTimeBudget do
23:    if π = nil or replan = true then
24:       {create a new plan}
25:       π ← CreatePlan()
26:       if π = nil then
27:          continue
28:       end if
29:    end if
       {get next plan task}
30:    t ← next task in π
31:    if t valid then
32:       decisionMade ← true {proceed with MCTS}
33:       f ← f_t {evaluation function of t}
34:    else {t invalid}
35:       π ← nil {continue}
36:    end if
37: end while
38: return MCTS(f)
```

the algorithm proceeds with the loop in lines 22 - 37. As long as it is within the given time budget, it tries to create a plan. When a plan π is found, its first task becomes the current task to execute in line 30. In the next step, the preconditions of this task are checked in line 31 ensuring the validity of the task given the current game state. If the task is invalid, the whole plan π is invalidated in line 35 leading to a re-planning. Otherwise, the task's evaluation function f is selected in line 33 and forwarded to the MCTS algorithm in line 38. MCTS then uses the remaining time budget for searching and returning an optimal player action.

The next time that any unit can execute an action, the algorithm first checks whether the current task t has been reached (by checking its post-conditions in line 14) and, if

it has not, whether it is still valid given the current game state. If the task is still valid and has not been reached, there is no need to change anything and we can continue using the current evaluation function (line 17). If the task has been reached, we can proceed with the next task in the plan jumping to line 22 (and potentially re-planning). However, if the task t has not been reached in line 14 but has become invalid given the current game state (similarly to failure detection in (Gonzlez Dorado et al. 2018; Weber, Mateas, and Jhala 2010)), a re-planning is triggered jumping from line 19 to line 22.

5 Experiments

5.1 Experiment Setup

Aiming to test the hybrid approach in its performance in microRTS and specifically the combination of strategic and tactic behavior in comparison to an agent that only incorporates tactical decision-making, we have compared our agent with the pure *naiveMCTS* agent (Ontañón 2013). We have used exactly the same MCTS parameters for our agent's MCTS part and the opponent. The parameters were the default parameters of the naiveMCTS agent: ϵ-greedy policies for π_0, π_l and π_g with $\epsilon_0 = 0.4$, $\epsilon_l = 0.3$ and $\epsilon_g = 0$ (see Section 2.2). The naiveMCTS agent used the *SqrtEF* evaluation function described in (Ontañón 2013) summing the resource cost of all player units weighted by the square root of the fraction of hit-points left and then subtracting the same sum for the opponent player. The maximal tree depth for MCTS was set to 10, the maximal simulation time to 100 frames, and the playout policy was set to *RandomBiasedAI* – an agent that is provided with the microRTS framework. This agent's actions are biased in the way that non-movement actions (collect, attack and return a resource) have a higher probability to be selected than movement actions.

For the strategical level of our agent we have created a simple HTN with 7 compound tasks, 16 methods, and the following 4 primitive tasks: *CollectRecources, BuildAndDefend, PreventAttack, AttackOpponent*. With this HTN, in the beginning of a game match the planner would usually create the following task sequence: *CollectRecources, BuildAndDefend, AttackOpponent*. However, depending on the map size and the units/buildings available at the start, it could create a plan without the *BuildAndDefend* task. Then, during the execution, it could detect an opponent's attack and re-plan scheduling the *PreventAttack* task first and then continuing with other tasks.

With these HTN tasks, the agent would assign one of the 4 corresponding evaluation functions (*CollectEF, BuildEF, PreventEF, AttackEF*) to its MCTS. Some functions took the same variables into account, however the weights used for the weighted sum were different. As already mentioned in Section 4.1, we have tweaked the weights manually according to our game knowledge. We have used the same evaluation functions for all map sizes tested in our experiments. Monitoring the progress of the high-level tasks and the environment the agent would decide when to switch between the tasks and thus switch between evaluation functions.

In addition to directly comparing our agent with the naiveMCTS agent, we tested both agents' performance against further agents. We selected the *AHTN* agent (Adversarial hierarchical Task Network) (Ontañón and Buro 2015) that comes with a predefined HTN, and the winner of 2017 microRTS AI competition – *StrategyTactics* (Barriga, Stanescu, and Buro 2017). We have also tested both agents against the baseline agent *RandomBiasedAI*. However, since both agents won all games against this agent, we did not put these results into our analysis.

Testing our agent against the 3 opponents (and naiveMCTS against 2 opponents), we have run 50 games on each map (with 25 games on each player side) for each pair. Therefor, we used 3 maps of the size 8×8 cells, 2 maps of the size 16×16 cells, and 2 maps of the size 24×24 cells. The maximum computation time of each agent's move was limited to 100 milliseconds (frametime). Following the official rules of the microRTS AI Competition (Ontañón 2017b), we have run the matches on the small maps for maximum 3000 frames, on the mid-size maps for maximum 4000 frames, and on the bigger maps for maximum 5000 frames. After this time a match was considered a tie.

5.2 Results

After manually tweaking the weights in the evaluation functions, our agent (HTN-MCTS) was able to show behaviors very close to the ones expected when performing the distinct HTN tasks. The interim results have shown that the hybrid agent was able to either win or achieve a tie in the majority of matches playing against the original naiveMCTS agent on all map sizes as shown in Figures 2 – 4.

Furthermore, the experiments have shown how much impact the weights have on the agent's behavior and how it is possible to achieve this behavior without giving the agent any explicit commands. Figure 1 shows the progress of a match played by the hybrid agent (striped units) against the naiveMCTS agent on the *NoWhereToRun9x8* map. (The blue, red, and violet cell background colors tell whether a cell would be visible for our agent, the opponent or both respectively if the game was played in the partially observable mode. However, these are not important for our experiments since we test the agent with full observability first.) In the first step, the hybrid agent was executing the HTN task *CollectRecources*. The task's evaluation function was taking into consideration the distance of each worker to the closest resource or the basis depending on whether or not he was carrying a resource. Therefore, all units stayed within a small radius of the base while the opponent's units spread around the available space.

In the next step, having enough worker units and resources, the agent could not attack the opponent because of the resource wall in the middle of the map. Therefore, the HTN planner created a new high-level plan scheduling the *BuildandDefend* task to be executed until meeting the attack conditions. In the second part of Figure 1, the agent created light military units which defended the base while preparing an attack. Again, this was done implicitly by maximizing the number of military units and minimizing their distances to the agent's base. At the same time, the opponent was cre-

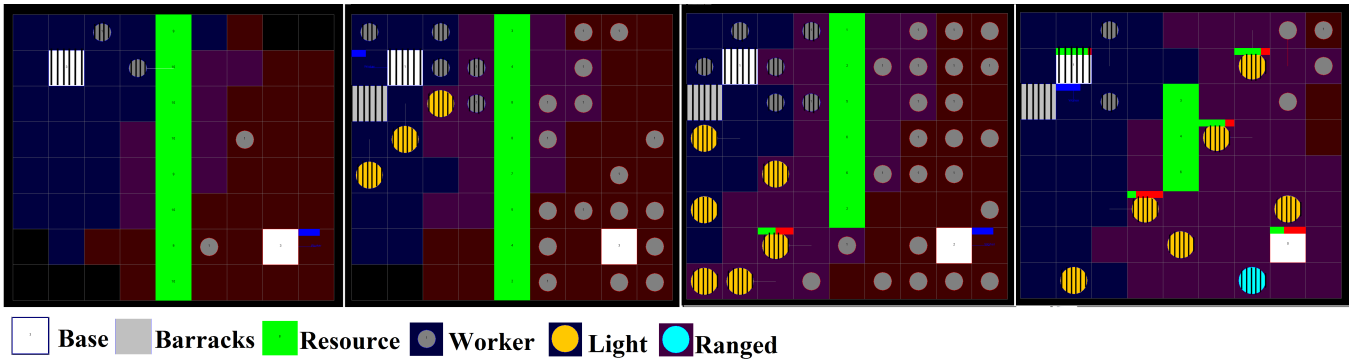


Figure 1: Progress of a match between our agent (striped units) and naiveMCTS.

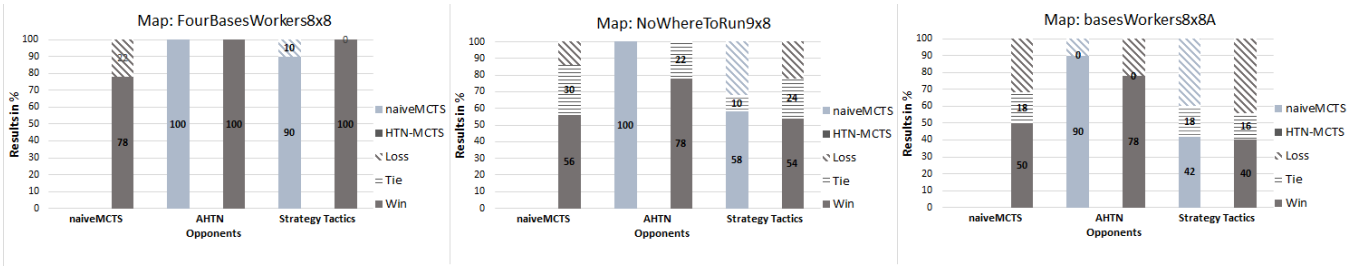


Figure 2: Results of 50 matches played by our agent (HTN-MCTS) and naiveMCTS against 3 (2) opponents on small maps.

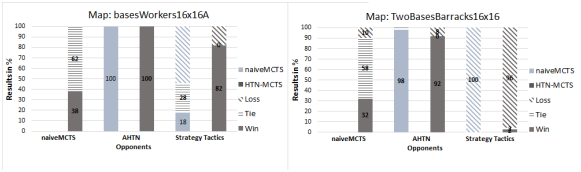


Figure 3: Results of 50 matches played by our agent (HTN-MCTS) and naiveMCTS against 3 (2) opponents on mid-size maps.

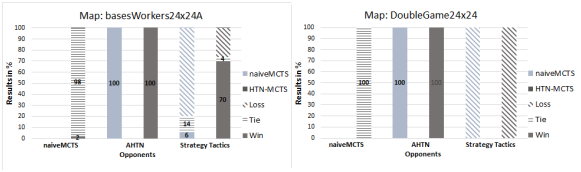


Figure 4: Results of 50 matches played by our agent (HTN-MCTS) and naiveMCTS against 3 (2) opponents on bigger maps.

ating more worker units since his function was maximizing the health points of units in general (not specifically military units) and creating workers was the cheapest in regards to resource consumption.

Once the wall in the middle was opened, the opponent's units spread further into our agent's part of the map. When they came too close to our agent's base, an attack was detected by the monitoring system and the HTN planner re-planned scheduling a *PreventAttack* task first. Thus, the agent stayed reactive to changes in the environment. Now, this task's evaluation function was minimizing the distance of every unit to the opponent unit closest to the base (red worker in bottom part of the map) and minimizing its health points while maximizing the health points of friendly units. Part 3 of Figure 1 shows the movement of the units towards the opponent worker.

Finally, when the preconditions for the attack task were reached, our agent started the *AttackOpponent* task minimizing the distance to the opponent's base and units as well as their health points while maximizing the health points of friendly units. Only in this step did the agent's units start visibly moving towards the other end of the map and following the opponent's units as shown in the last part of Figure 1.

In general, these experiments have shown that defining an abstract high-level task by evaluation functions for MCTS can lead to interesting and visibly distinct emergent behaviors. However, they have also shown that the balance between weights has a big impact on the behaviors. For example, we could see in most games on the mid-size and bigger maps that our agent was not *aggressive* enough when attacking the opponent with military units although the attack behavior was aggressive on small maps when the attack was performed by workers mostly. For that reason, most games between our agent and naiveMCTS on mid-size and bigger maps resulted in a tie as shown in Figures 3 and 4. In these cases, our agent would not destroy all of the opponent's units while strongly defending friendly units and not letting the opponent win.

We assume that, for one part, this difference in aggressiveness was due to the different weighting of health points of different unit types. Thus, destroying an enemy unit might have been too costly while risking to lose a friendly unit. An additional reason could be weights assigned to distance minimization in relation to the weights of the functions responsible for health points optimization. Thus, in smaller maps, it could have been more optimal for the agent to reach the opponent's base than to keep friendly units alive while it was the opposite way on mid-size maps.

The same defensive behavior was visible when playing against the agents AHTN and StrategyTactics, both of which are very aggressive and purposefully attack their opponents (as opposed to naiveMCTS). A match against one of these agents usually progressed as follows: our agent started building barracks and military units while the opponent quickly started an attack. That way, our agent had to stop expanding its army and executed the *PreventAttack* task for as long as needed. The difference between AHTN and StrategyTactics was, however, that the former agent fully concentrated on the attack sending all of his units towards our base. That allowed our agent to destroy all enemy units without having to move across the map and balance between distance and hit points. This way, our agent was able to win most games against AHTN on all map sizes.

In contrast, StrategyTactics continued to expand its army while performing an assault with only a part of its units. Depending on how well/fast our agent was able to destroy the incoming enemy units and return to the build task, it was either able to start an attack itself and destroy the remaining units as shown in the following video[2] or was defeated. For that reason, out agent outperformed the naiveMCTS agent when playing against StrategyTactics even on the mid-size map *basesWorkers16x16A* (Figure 3) and *basesWorkers24x24A* (Figure 4).

However, our agent was not able to win against StrategyTactics on the maps *TwoBasesBarracks16x16* (Figure 3) and *DoubleGame24x24* (Figure 4) which were different from other maps. Here, each player started with 2 bases (and 2 barracks) and more resources. That gave the StrategyTactics agent an even better chance to start a quick assault while our agent was trying to fully build up his army before attacking. We assume that in order to improve our agent's performance on such special maps, we need to make changes to the high-level HTN, allowing for quick attacks under different preconditions.

Finally, the experiments have shown that the proposed monitoring approach allows for a combination of long-term planning and reactive execution. Especially in games against AHTN and StrategyTactics, our agent was able to detect changes in the environment (when the opponents were performing an attack) and trigger re-planning on a high-level.

6 Conclusions and Future Work

In this work we have introduced a planning and execution approach which uses an HTN planner for high-level planning and MCTS for micro-management of multiple units. Instead of explicitly defining *how* to execute HTN tasks, we define evaluation functions telling *what* needs to be optimized when executing a task. An evaluation function for a task is a weighted sum of functions optimizing, for example, distances, health points or resources. These evaluation functions are then used by MCTS selecting an optimal distribution of unit actions. The game environment and the progress of the high-level plan are monitored during execution and, if required, re-planning is triggered on the higher level.

Our first experiments in the microRTS environment have shown that this combination of the two planning approaches can lead to visible emergent behaviors. Furthermore, the interleaved planning, monitoring and execution has allowed the agent to stay reactive while following the high-level tasks. Additionally, even with a very simple HTN, the hybrid agent was able to perform similarly or better than a pure MCTS agent that was using a single evaluation function throughout the whole game match. It was also able to outperform a pure HTN agent and performed well against a strong agent that combined strategical and tactical behaviors.

The major difficulty in creating planning domains for this hybrid approach has been proven to be the balancing of weights used in the weighted sums of evaluation functions. Tweaking these weights has a big impact on the units' behaviors. For that reason, our first step towards future work is finding an appropriate way to automatically learn these weights. Alternatively, we might use a different approach for multi-objective optimization (Perez et al. 2015) instead of the weighted sums. Also, the evaluation functions might be more complex, for example controlling the distances between all units allowing for swarms with formations. Additionally, we might experiment with different high-level HTNs either creating more detailed networks or tweaking the pre- and post-conditions of the current one. Finally, this approach could be tested in an even more complex environment such as the full version of StarCraft or the partially observable mode of microRTS.

[2]Experiment video: `https://youtu.be/Eox_ab836tM`

References

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine learning* 47(2-3):235–256.

Balla, R.-K., and Fern, A. 2009. UCT for tactical assault planning in real-time strategy games. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, 40–45.

Barriga, N. A.; Stanescu, M.; and Buro, M. 2015. Puppet search: Enhancing scripted behavior by look-ahead search with applications to real-time strategy games. In *Proceedings of the 11th Artificial Intelligence and Interactive Digital Entertainment Conference*, 9–15.

Barriga, N. A.; Stanescu, M.; and Buro, M. 2017. Combining strategic learning and tactical search in real-time strategy games. *arXiv preprint arXiv:1709.03480*.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of monte

carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4(1):1–43.

Chung, M.; Buro, M.; and Schaeffer, J. 2005. Monte carlo planning in RTS games. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 117–124.

Churchill, D., and Buro, M. 2013. Portfolio greedy search and simulation for large-scale combat in starcraft. In *Proceedings of the IEEE Conference on Computational Intelligence in Games*. IEEE.

Gonzlez Dorado, J. C.; Veloso, M.; Fernndez, F.; and Garca-Olaya, A. 2018. Task monitoring and rescheduling for opportunity and failure management. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling. Workshop on Integrated Planning, Acting and Execution.*, 24–31.

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2018a. Plan and goal recognition as HTN planning. In *IEEE 30th International Conference on Tools with Artificial Intelligence*, 466–473. IEEE.

Höller, D.; Bercher, P.; Behnke, G.; and Biundo, S. 2018b. HTN plan repair using unmodified planning systems. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling. Workshop on Hierarchical Planning*, 26–30.

Ishihara, M.; Miyazaki, T.; Chu, C. Y.; Harada, T.; and Thawonmas, R. 2016. Applying and improving monte-carlo tree search in a fighting game AI. In *Proceedings of the 13th International Conference on Advances in Computer Entertainment Technology*, 27–32. ACM.

Justesen, N.; Tillman, B.; Togelius, J.; and Risi, S. 2014. Script-and cluster-based UCT for starcraft. In *Proceedings of the IEEE Conference on Computational Intelligence in Games*. IEEE.

Moraes, R. O.; Marino, J. R.; Lelis, L. H.; and Nascimento, M. A. 2018. Action abstractions for combinatorial multi-armed bandit tree search. In *Proceedings of the 14th Artificial Intelligence and Interactive Digital Entertainment Conference*, 74–80.

Nau, D.; Cao, Y.; Lotem, A.; and Munoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*, 968–973. Morgan Kaufmann Publishers Inc.

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *Journal of artificial intelligence research* 20:379–404.

Ontañón, S., and Buro, M. 2015. Adversarial hierarchical-task network planning for complex real-time games. In *Proceedings of the 24th International Conference on Artificial Intelligence*, 1652–1658. AAAI Press.

Ontañón, S.; Barriga, N. A.; Silva, C. R.; Moraes, R. O.; and Lelis, L. H. 2018. The first microrts artificial intelligence competition. *AI Magazine* 39(1).

Ontañón, S. 2013. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In *Proceedings of the 9th Artificial Intelligence and Interactive Digital Entertainment Conference*, 58–64.

Ontanón, S. 2016. Informed monte carlo tree search for real-time strategy games. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*, 1–8. IEEE.

Ontanón, S. 2017a. Combinatorial multi-armed bandits for real-time strategy games. *Journal of Artificial Intelligence Research* 58:665–702.

Ontañón, S. 2017b. MicroRTS AI Competition, https://sites.google.com/site/micrortsaicompetition/.

Orkin, J. 2006. Three states and a plan: the AI of FEAR. In *Game Developers Conference*.

Perez, D.; Mostaghim, S.; Samothrakis, S.; and Lucas, S. M. 2015. Multiobjective monte carlo tree search for real-time games. *IEEE Transactions on Computational Intelligence and AI in Games* 7(4):347–360.

Perez, D.; Rohlfshagen, P.; and Lucas, S. M. 2012. Monte-carlo tree search for the physical travelling salesman problem. In *European Conference on the Applications of Evolutionary Computation*, 255–264. Springer.

Pozanco, A.; Yolanda, E.; Fernández, S.; and Borrajo, D. 2018. Counterplanning using goal recognition and landmarks. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling. Workshop on Distributed and Multi-Agent Planning.*, 17–24.

Sailer, F.; Buro, M.; and Lanctot, M. 2007. Adversarial planning through strategy simulation. In *IEEE Symposium on Computational Intelligence and Games*, 80–87. IEEE.

Sironi, C. F.; Liu, J.; Perez-Liebana, D.; Gaina, R. D.; Bravi, I.; Lucas, S. M.; and Winands, M. H. 2018. Self-adaptive MCTS for general video game playing. In *International Conference on the Applications of Evolutionary Computation*, 358–375. Springer.

Soemers, D. 2014. Tactical planning using MCTS in the game of starcraft. Bachelors thesis, Department of Knowledge Engineering, Maastricht University.

Stanescu, M.; Barriga, N. A.; and Buro, M. 2014. Hierarchical adversarial search applied to real-time strategy games. In *Proceedings of the 10th Artificial Intelligence and Interactive Digital Entertainment Conference*, 66–72.

Uriarte, A., and Ontañón, S. 2014. Game-tree search over high-level game states in RTS games. In *Proceedings of the 10th Artificial Intelligence and Interactive Digital Entertainment Conference*, 73–79.

Weber, B. G.; Mateas, M.; and Jhala, A. 2010. Applying goal-driven autonomy to starcraft. In *Proceedings of the 6th Artificial Intelligence and Interactive Digital Entertainment Conference*, 101–106.

Interleaving Acting and Planning Using Operational Models

Sunandita Patra[1], **Malik Ghallab**[2], **Dana Nau**[1], **Paolo Traverso**[3]
patras@cs.umd.edu, malik@laas.fr, nau@cs.umd.edu, traverso@fbk.eu
[1]Department of Computer Science and Institute for Systems Research, University of Maryland, College Park, USA
[2]Centre national de la recherche scientifique (CNRS), Toulouse, France
[3]Fondazione Bruno Kessler (FBK), Povo - Trento, Italy

Abstract

In (Patra et al. 2019) we proposed and implemented a framework for planning with *operational models*, i.e., models that describe how to perform actions, with rich control structures for closed-loop online decision-making. As described in (Patra et al. 2019), the acting component RAE, inspired by the well-known PRS system, calls the planner RAEplan, which plans by doing Monte Carlo rollout simulations of the actor's operational models.

In this paper, we show how this framework can be used to interleave acting and planning with operational models in different ways. We extend the acting component RAE with heuristics to decide when and how to call the planning component RAEplan. This allows us to realize more or less reactive behaviors. For instance, the acting component RAE may decide to call the planner just when it fails or anytime a decision needs to be made. Moreover, RAE can decide whether to bound the depth of the search during planning, and whether to do acting and planing concurrently. The planning algorithm in this paper takes into consideration the depth of the search. We call the modified planning algorithm RAEplan-LookAhead. We implement the RAEplan-LookAhead algorithm and do its experimental evaluation on a simulated domain called Search and Rescue.

Introduction

Several approaches for the integration of planning, acting, and execution have been proposed so far, see, e.g., (Vaquero et al. 2018). Some of them (e.g., lookahead methods, see e.g., (Ghallab, Nau, and Traverso 2016) for a survey) are based on the idea of generating a partial plan, for example the next few "good" actions, and then acting, i.e., performing all or some of the generated actions, and repeating these two steps from the state that has been reached. In this way, the planner knows exactly which of the many possible states of the world has actually been reached, and the uncertainty as well as the search space is significantly reduced. Moreover, interleaving planning and acting provides the ability to deal with dynamic environments and exogenous events.

Most of the previous approaches to interleaving planning and execution perform planning with *descriptive models*, which represent actions at a rather abstract level, e.g., with preconditions and effects. This representation is tailored to efficiently compute, given some conditions on state variables (the action preconditions), how the values of the state variables change (the action effects). However, when planning needs to be interleaved with acting, most of the works highly underestimate the problem of mapping descriptive models to and from *operational models*, which describe *how* to perform actions, with rich control structures for closed-loop online decision-making. The mapping between the descriptive model and the operational model is often given for granted, simply assuming that the acting/execution mechanism returns the actual state (the values of state variables) at the abstract level in which the agent can start to do planning again.

In this paper we take a different approach. We build upon our work presented in (Patra et al. 2019) (see also (Patra et al. 2018)), in which we propose to use a single representation, the operational model, for both acting and planning, and to do planning by reasoning directly with the actor's operational models. In our approach, the agent does not start from planning and then calls the execution platform when needed. We rather start the other way around. The acting component RAE, inspired by the well-known PRS system, calls the planner RAEplan, which plans by doing Monte Carlo rollout simulations of the actor's operational models. RAE uses a hierarchical task-oriented operational representation. A collection of refinement *methods* describes alternative ways to handle *tasks* and react to *events*. RAE calls RAEplan to decide how to refine tasks or events. RAEplan does Monte Carlo rollouts with applicable refinement methods.

In this paper, we extend this framework and show how it can be used to interleave acting and planning in different ways. We extend the acting component RAE with heuristics to decide when an how to call the planning component RAEplan. This allows us to realize more or less reactive behaviours. For instance, the acting component RAE can decide to call the planner just when it fails or anytime a decision must be made, i.e., anytime one upon different applicable methods must be selected. Moreover, our extended planning algorithm RAEplan-LookAhead, when called by RAE, can decide whether to complete the Monte Carlo rollout, or to bound the depth of the search according to different heuristics, e.g., to save time. Finally, RAE and RAEplan-

LookAhead can be run concurrently and interact in different ways.

We implement different techniques for interleaving acting and planning using heuristics and evaluate them on the Search and Rescue domain. With this experimental evaluation, we show the benefits of interleaving acting and planning with operational models.

The paper is structured as follows. In the next section, we first provide some background on the key concepts described in (Patra et al. 2019) to keep the paper self-contained. Following that, we describe different techniques for the interleaving of acting and planning. We then describe the operational models for the Search and Rescue domain, and provide an experimental evaluation. We finally discuss the related and future work.

Background

In this section, we briefly review the key elements of the approach presented in (Patra et al. 2019), to make the paper self-contained. For the details of the acting component RAE and of the planning component RAEplan, please refer to (Patra et al. 2019).

RAE (Refinement Acting Engine) is from (Ghallab, Nau, and Traverso 2016, Chapter 3). It is is based on a hierarchical task-oriented operational representation with an expressive, general-purpose language offering rich control structures for closed-loop online decision-making. A collection of refinement methods describes 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 *subtasks*, which need to be refined recursively, and sensory-motor *commands*, which query and change the world non-deterministically. Notice that we assume that *methods, tasks, and subtasks are manually programmed*. We believe this assumption is the practical way to build agents that can behave reactively and deal with realistic and complex applications.

RAE implements a reactive system. At each loop, it gets in input a task or event that comes in from an external source, such as the user or the execution platform, and it creates a *refinement stack*, analogous to a computer program's execution stack. An agenda keeps the set of all current refinement stacks.

Task frames and refinement stacks. A *task frame* is a four-tuple $r = (\tau, m, i, \textit{tried})$, where τ is a task, m is the method instance used to refine τ, i is the current instruction in body(m), with $i = \mathsf{nil}$ if we haven't yet started executing body(m), and *tried* is the set of methods that have been already tried and failed for accomplishing τ.

A *refinement stack* is a finite sequence of stack frames $\textit{stack} = \langle \rho_1, \ldots, \rho_n \rangle$. If *stack* is nonempty, then top(*stack*) $= \rho_1$; rest(*stack*) $= \langle \rho_2, \ldots, \rho_n \rangle$; *stack* $=$ top(*stack*).rest(*stack*). To denote pushing ρ onto *stack*, we write $\rho.\textit{stack} = \langle \rho, \rho_1, \rho_2, \ldots, \rho_n \rangle$. Refinement stacks used during planning will have the same semantics, but we will use the notation *stack* instead of *stack* to distinguish it from the acting stack.

When an execution failure occurs with a method instance, then RAE calls a Retry procedure. Retry tries another applicable method instance that it hasn't tried already. Notice that when a Retry is called, the failed method has already been partially executed; it has changed the current 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. We call *retry ratio* the number of times that RAE had to call the Retry procedure, divided by the total number of tasks to accomplish.

Rather than behaving purely reactively, the agent interleaves acting with planning to decide how to refine tasks or events. Planning is performed by Monte Carlo rollouts with applicable refinement methods. Planning is therefore performed with all the same constructs and operations of the operational model used to act, all but a simulation of commands. Commands are indeed simulated when planning and performed by an execution platform in the real world when acting. During planning, when a refinement method contains a command, the planner takes samples of its possible outcomes, using either a domain-dependent generative simulator, when available, or a probability distribution of its possible outcomes.

RAEplan does a recursive search to optimize a criterion. It chooses a refinement method that has a refinement tree with a minimum expected cost for accomplishing a task (along with the remaining partially accomplished tasks in the current refinement stack). It minimizes the expected cost, i.e., the expected cost of the plan for accomplishing all the tasks in the refinement stack. In order to take into account possible failures, which would have an infinite cost, cost minimization is done by maximizing an *efficiency* criteria, which is the reciprocal of the cost.

Efficiency. We define the *efficiency* of accomplishing a task to be the reciprocal of the cost. Let a decomposition of a task τ have two subtasks, τ_1 and τ_2, with cost c_1 and c_2 respectively. The efficiency of τ_1 is $e_1 = 1/c_1$ and the efficiency of τ_2 is $e_2 = 1/c_2$. The cost of accomplishing both tasks is $c_1 + c_2$, so the efficiency of accomplishing τ is

$$1/(c_1 + c_2) = e_1 e_2/(e_1 + e_2). \tag{1}$$

If $c_1 = 0$, the efficiency for both tasks is e_2; likewise for $c_2 = 0$. Thus, the incremental efficiency composition is:

$$\begin{aligned} e_1 \bullet e_2 = e_2 \text{ if } e_1 = \infty, \text{ else} \\ e_1 \text{ if } e_2 = \infty, \text{ else } e_1 e_2/(e_1 + e_2). \end{aligned} \tag{2}$$

If τ_1 (or τ_2) fails, then c_1 is ∞, $e_1 = 0$. Thus $e_1 \bullet e_2 = 0$, meaning that τ fails with this decomposition. Note that formula 2 is associative.

Moreover, RAEplan has two parameters b and k. Parameter b denotes how many different method instances to examine for each task. Parameter k denotes how large a sample size must be for each command. The estimated efficiency $E^*_{b,k}(s, \textit{stack})$ calculated in a given state s for a refinement stack *stack* depends on both b and k. The larger the values of b and k in $E^*_{b,k}$, the more plans RAEplan will examine. In the

planning algorithm proposed in this paper called RAEplan-LookAhead, we add an additional sub-script called d to efficiency in order to bound the depth of Monte Carlo rollouts that RAEplan-LookAhead examine. In RAEplan, d is always infinite. As $d \to \infty$, the behavior of RAEplan-LookAhead becomes similar to RAEplan. Please see the next section for details.

Interleaving Acting and Planning

In (Patra et al. 2019), the acting algorithm RAE always waits for the planner RAEplan to complete its search and return a refinement method for a task τ. The time taken by RAEplan to complete the search increases with the increase in size of the refinement tree for τ. However, because the planning is happening online, this may create a long wait time before the actor takes an action. We can have several strategies to reduce the wait time of the actor and discuss them below.

Strategy 1: Active Planning. This is the simplest case where RAE calls RAEplan every time it needs to refine a task τ. RAE waits for RAEplan to complete its search and refines τ according to what RAEplan suggested. This strategy is implemented in the paper (Patra et al. 2019). The advantage of this approach is that RAEplan returns the best possible suggestion for given values of b and k. The disadvantage is that the actor RAE need to wait until RAEplan returns. It has no control over the wait time.

Strategy 2: Include Heuristics. The idea is similar to a lookahead search. When RAEplan searches for the most efficient method for a task τ, it does several Monte Carlo rollouts. Every such rollout corresponds to a complete refinement tree for τ. Our idea is that instead of looking at complete rollouts like RAEplan, RAEplan-LookAhead only rolls out upto depth d. When the length of the rollout reaches depth d, we estimate the efficiency of the remaining part of the rollout using a heuristic function. The heuristic may be domain dependent or domain independent. We call the modified algorithm RAEplan-LookAhead. The pseudocode is as follows:

RAEplan-LookAhead$(s, \tau, \mathit{tried}, \mathit{stack}, d)$
  $M \leftarrow \mathit{Candidates}(\tau, s) \setminus \mathit{tried}$
  if $d = 0$ then return $M[1]$
  else
    $m_{opt} \leftarrow \text{argmax}_{m \in M} E^*_{b,k,d-1}(s, (\tau, m, 0, \mathit{tried}).\mathit{stack})$
    if $m_{opt} =$ None then return $M[1]$
    else return m_{opt}

Above, s is the current state, *tried* is the set of refinement method instances which has been tried by RAE to accomplish τ and failed. *stack* is the current refinement stack. $\mathit{Candidates}(\tau, s)$ is the set of applicable method instances for τ in current state s. d is the maximum search depth. Note that $d = 0$ corresponds to the situation where RAE acts purely reactively and no planning is done. RAEplan-LookAhead optimizes a criterion called expected efficiency which is based on the definition in the previous section with an additional parameter d (in subscript). The expected efficiency is calculated depending on whether the current step is a task or command and also the current depth d. The definition of E^* is recursive and the value of d decreases by one at every recursive call.

Estimated efficiency. We now define $E^*_{b,k,d}(s, \mathit{stack})$ as an estimate of expected efficiency of the optimal plan for the tasks in stack *stack* when the current state is s. The parameters b and k denote, respectively, how many different method instances to examine for each task, and how large a sample size to use for each command. d denotes how much further RAEplan-LookAhead is allowed to search.

If *stack* is empty, then $E^*_{b,k,d}(s, \mathit{stack}) = \infty$ because there are no tasks to accomplish. Otherwise, let $(\tau, m, i, \mathit{tried}) = \text{top}(\mathit{stack})$. Then $E^*_{b,k,d}(s, \mathit{stack})$ depends on whether i is a command, an assignment statement, or a task and whether the current depth d is greater than 0:

- If i is a command and $d > 0$, then $E^*_{b,k,d}(s, \mathit{stack}) =$

$$\tfrac{1}{k} \textstyle\sum_{s' \in S'} \frac{1}{\mathit{cost}(s,i,s')} \bullet E^*_{b,k,d-1}(s', \mathsf{next}(s', \mathit{stack})), \quad (3)$$

  where S' is a random sample of k outcomes of command i in state s, with duplicates allowed. $\mathsf{next}(s', \mathit{stack})$ is the refinement stack after performing command i taking into account the effect of control statements like if-else or loops. Since S' has the probability distributions of the outcomes of the commands, it converges asymptotically to the expected value of E^*.

- If i is a command and $d = 0$, then

$$E^*_{b,k,0}(s, \mathit{stack}) = \frac{1}{\text{Heuristic-Estimate}(s, \mathit{stack})} \quad (4)$$

- If i is an assignment statement, then $E^*_{b,k,d}(s, \mathit{stack}) = E^*_{b,k,d}(s', \mathsf{next}(s', \mathit{stack}))$, where s' is the state produced from s by performing the assignment statement.

- If i is a task and $d > 0$, then $E^*_{b,k,d}(s, \mathit{stack})$ recursively optimizes over the candidate method instances for i. That is:

$$E^*_{b,k,d}(s, \mathit{stack}) = \max_{m \in M'} E^*_{b,k,d-1}(s, (i, m, \mathsf{nil}, \emptyset).\mathit{stack}), \quad (5)$$

  where $M' = \mathit{Candidates}(i, s)$ if $|\mathit{Candidates}(i, s)| \leq b$, and otherwise M' is the first b method instances in the preference ordering for $\mathit{Candidates}(i, s)$.

- If i is a task and $d = 0$, then $E^*_{b,k,0}(s, \mathit{stack})$ is a heuristic estimate of accomplishing the remaining stack. That is:

$$E^*_{b,k,0}(s, \mathit{stack}) = \frac{1}{\text{Heuristic-Estimate}(s, \mathit{stack})}. \quad (6)$$

We have implemented RAEplan-LookAhead in this paper for a simulated domain called Search and Rescue with two domain dependent heuristics and various values of depth d. As d approaches infinity, the behavior of RAEplan-LookAhead should become similar to the RAEplan algorithm in (Patra et al. 2019). The results are presented in the Experimental Evaluation section.

Strategy 3: Lazy Planning. RAE calls RAEplan-LookAhead the first time it receives a task. RAEplan-LookAhead needs to be modified so that it returns the

most expected refinement tree because this strategy requires that we have a plan and not just one suggested refinement method instance. RAE executes the refinement tree via in-order traversal until it encounters a mismatch between the current state and the state expected from the refinement tree, or a command fails. The drawback of this approach is that in an environment with dynamic events, our plan might become too old and lead to dead-ends which we could have avoided if we used Strategy 1. The advantage of this approach is that it saves the time of computing a similar plan again in case their are no dynamic or exogenous events. However, if commands are non-deterministic, the chances of getting a similar plan are low and this is probably not a good approach.

Strategy 4: Concurrent Planning. RAEplan-LookAhead always runs in parallel to RAE and keeps track of the most recent plan for the current task at hand. As explained in the case of lazy Planning, RAEplan-LookAhead needs to be modified so that it returns a refinement tree instead of just a refinement method instance. Whenever RAE needs advice from RAEplan, RAEplan suggest the refinement method from its current refinement tree. One drawback of this strategy is that, the current refinement tree may not have taken into account the most recent dynamic or exogenous events of the environment. However, this should be better than the lazy strategy because the planner is never idle. This method can be useful when the actor has access to multiple cores and can do multi-tasking.

One could use any one of the above strategies or a combination of them depending on the domain and the nature of tasks that need to be accomplished.

Domain: Search and Rescue

Consider that some natural disaster has happened in a 2D area. People are trapped or injured at certain locations in this area which has no particular graph or map. UAVs continuously survey the area and find people who need help. The detection happens by capturing images via the front and bottom cameras that the UAVs are equipped with. The clarity of the image depends upon various weather conditions and the altitude at which the UAV is flying. We assume that a human expert or some computer vision algorithm identifies correctly whether a person needs help or not from the image. Once a person in need of help has been identified, the UAV transfers control to the UGVs operating on the ground. Ground locations are represented via integral coordinates. The UGVs navigate following certain patterns. In order to move from one location to another, UGVs may take a straight route, a curved route or a Manhattan route. There may or may not be obstacles in their path. If it finds an obstacle, it needs to take a different route to reach its destination. UAVs always fly from one location to the other via a straight route. They may fly in two different altitudes. UGVs are useful for transporting first-aid and medicine to doctors and volunteers or the person in need. First-aid and medicines can be picked up from the base camp or taken from other UGVs which have them. Once helper robot and/or human experts have reached the location of the injured person, they may not find the person immediately. They might need to do some sensing and searching, which can involve removing debris or looking around.

Example 1. *Consider a set R of robots performing search and rescue operations in a partially mapped area. The robots have to find persons in some area and leave them a package of supplies (medication, food, water, etc.). This domain is specified with state variables such as* $\mathsf{robotType}(r) \in \{\textit{UAV, UGV}\}$, $r \in R$; $\mathsf{hasSupply}(r) \in \{\top, \bot\}$; $\mathsf{loc}(r) \in L$, *for* $L = \{(x, y) |\ x \text{ and } y \text{ are integers}\} \cup \{\textit{BASE}\}$.

These robots can use commands such as $\textsc{Detect}(r, \textit{camera, class})$ *which detects if an object of some class appears in images acquired by camera of* r, $\textsc{TriggerAlarm}(r, l)$, $\textsc{DropSupply}(r, l)$, $\textsc{LoadSupply}(r, l)$, $\textsc{Takeoff}(r, l)$, $\textsc{Land}(r, l)$, $\textsc{MoveTo}(r, l)$, $\textsc{FlyTo}(r, l)$. *They can address tasks such as:* $\mathsf{search}(r, \textit{area})$, *which makes a UAV* r *survey in sequence the locations in area,* $\mathsf{survey}(r, l)$, $\mathsf{navigate}(r, l)$, $\mathsf{rescue}(r, l)$, $\mathsf{getSupplies}(r)$.

Here is a refinement method for the survey *task:*

$\mathsf{m1\text{-}survey}(r, l)$
  task: $\mathsf{survey}(r, l)$
  pre: $\mathsf{robotType}(r) = \text{UAV}$
  body: if $\textsc{Detect}$(r,"base-camera","person")=$\top$ then:
    if $\mathsf{hasSupply}(r)$ then $\mathsf{rescue}(r, l)$
    else $\textsc{TriggerAlarm}(r, l)$

This methods specifies that in the location l the UAV r detects if a person appears in the images from its base camera. In that case, it proceeds to a rescue task if it has supplies, otherwise it triggers an alarm event. This event is processed (by some other methods) by finding the closest robot not involved in a current rescue and assigning to it a rescue task for that location.

Here are two possible methods for the task $\mathsf{rescue}(r, l)$*:*

$\mathsf{m1\text{-}rescue}(r, l)$
  task: $\mathsf{rescue}(r, l)$
  pre: $\mathsf{robotType}(r) = \text{UAV}$
  body: if $\mathsf{hasSupply}(r)$ then
    if $\mathsf{loc}(r) = l$ then $\textsc{DropSupply}(r, l)$
    else do
      $\mathsf{navigate}(r, l)$
      $\mathsf{rescue}(r, l)$
  else do
    $\mathsf{navigate}(r, BASE)$
    $\textsc{LoadSupply}(r, BASE)$
    $\mathsf{rescue}(r, l)$

$\mathsf{m2\text{-}rescue}(r, p)$
  task: $\mathsf{rescue}(r, p)$
  pre: $(\mathsf{robotType}(r) = \text{UGV}) \wedge \mathsf{hasSupply}(r)$
  body: if $\mathsf{loc}(r) = l$ then $\textsc{DropSupply}(r, l)$
    else do
      $\mathsf{navigate}(r, l)$
      $\mathsf{rescue}(r, l)$

Note that the above methods are recursive. □

Experimental Evaluation

For our experiments, we generated 96 problems for the search and rescue domain randomly. Every problem has one incoming task, 'survey' or 'rescue' which arrives at a randomly chosen time in RAE's input stream. A problem may

have one to four robots (consisting of UAVs and UGVs). The location of a robot consist of its x and y coordinates in a 2D area. x and y are chosen to be random integers from the range [5, 30]. The location from where a person needs to be rescued is also generated randomly in the same way. Because our commands are nondeterministic, every problem with a particular combination of parameters of RAEplan-LookAhead is run 20 times. The experiments are run on a 2.6 GHz Intel Core i5 processor.

RAEplan-LookAhead has three main parameters: b, k and d. We first did experiments by changing b and k with d set to ∞. We measured the performance using three different metrics: efficiency, success ratio and retry ratio. It is not easy to measure the performance of an integrated planning and acting system. These three metrics were developed after much thought and details can be found in (Patra et al. 2019). Efficiency is the same discussed in the previous section. Success ratio is the number of successful jobs divided by the total number of incoming jobs. A *job* is a task that arrives in the input stream of RAE and does not include the subtasks generated from it. Retry ratio is the number of times RAE retries a task before succeeding (details can be found in the Section *Background*). The results for efficiency, success ratio and retry ratio are shown in Figures 1, 2 and 3 respectively.

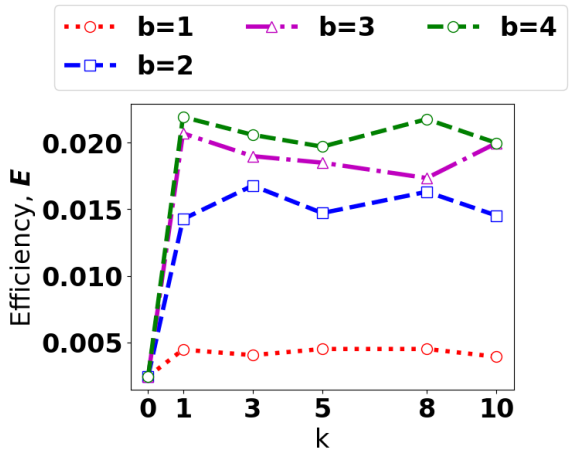


Figure 1: Efficiency E for various values of b and k in the Search and Rescue Domain. d is set to infinity.

From the plots of efficiency, success ratio and retry ratio, we observe that $k = 3$ is the most optimal value of k. Now, we fix k to the value 3 and do experiments by varying the parameter depth d of RAEplan-LookAhead. We do this with the following two heuristics:

1. **Zero Heuristic**: Heuristic is always 0.
2. **Distance Heuristic**: Heuristic is the distance of the agent trusted with the rescue operation and the location where the rescue operation needs to be performed.

We choose the value of depth d to be all values from the set $\{0, 3, 6, 9, 12, 15\}$. We also choose b to be all values from $\{1, 2, 3, 4\}$ because there can be a maximum of four method

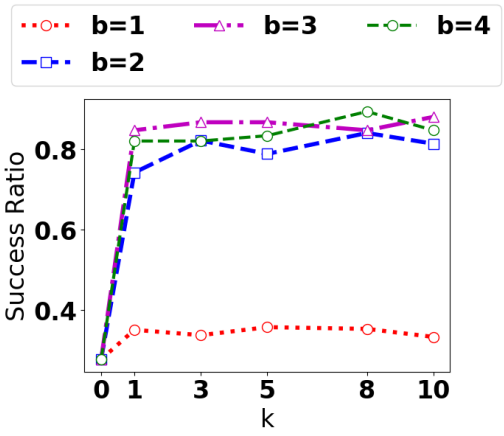


Figure 2: Success ratio (number of successful jobs/ total number of jobs) for various values of b and k in the Search and Rescue Domain.

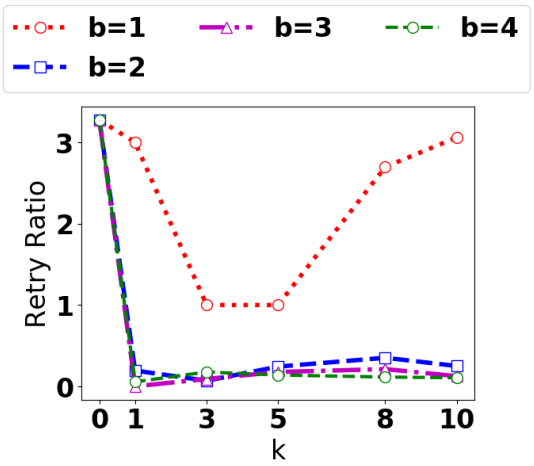


Figure 3: Retry Ratio (number of retries of RAE/ total number of jobs) for various values of b and k in the Search and Rescue Domain.

instances for any task in this domain. We set k to 3 as explained before.

Experiments with Zero Heuristic

We expect to see an improvement in efficiency with increase in depth d because the plans will be more accurate if we examine rollouts till a higher depth before using a heuristic estimate. It is indeed the case as observed in Figure 4. The success ratio also increases for the same reason as observed in Figure 5.

It is interesting to see that the retry ratio decreases upto depth $d = 6$ but then starts increasing with increase in d. In general, we would expect the retry ratio to decrease with increase in d because RAE should be able to accomplish tasks with fewer attempts when plans are more accurate. The increase in efficiency and success ratio confirms that. However, note that retry ratio is measured and compared only for

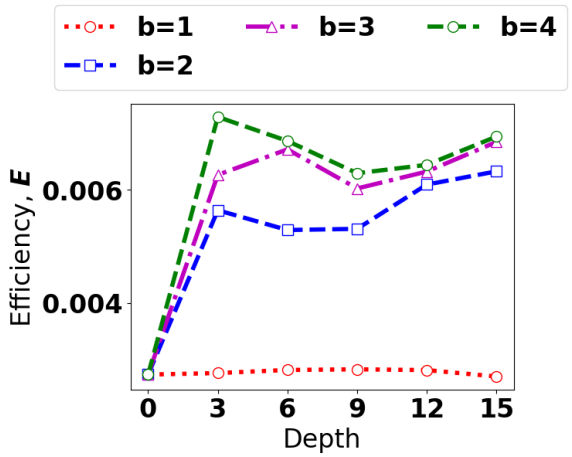


Figure 4: Efficiency E for various values of b and depth d in the Search and Rescue Domain for Zero heuristic.

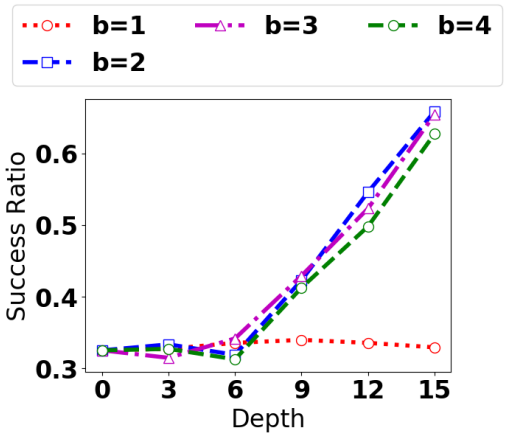


Figure 5: Success ratio (number of successful jobs/ total number of jobs) for various values of b and depth d in the Search and Rescue Domain for Zero heuristic.

the successful jobs, jobs that succeed for all values of b and d because it will be unfair to compare the retries of a failed job to the retries of a successful job. For this set of sub-problems in the Search and Rescue domain, it is possible that for some sub-task, RAEplan-LookAhead finds a method which is more efficient but not very reliable to succeed. The failure is not very dangerous because the success ratio does not suffer as seen in Figure 5.

Experiments with Distance Heuristic

Like in the case of Zero heuristic, the experiments with Distance heuristic also show that the efficiency increases with increase in b and depth d. This can be seen in Figure 7. The success ratio also increases with increase in b and d as seen in Figure 8. The behavior of retry ratio shown in Figure 9 is similar to that of zero heuristic. It decreases upto $d = 6$ and then increases. We believe the reason for this behavior is same as discussed in the case of zero heuristic.

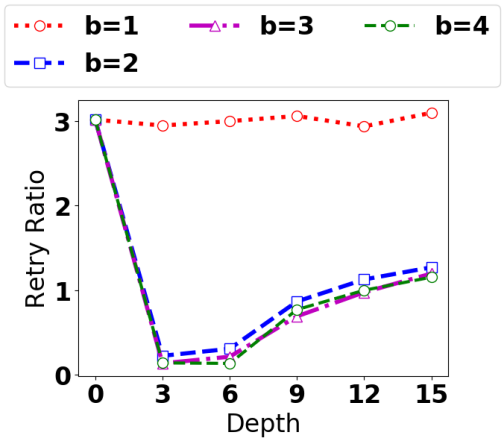


Figure 6: Retry ratio for various values of b and depth d in the Search and Rescue Domain for Zero heuristic.

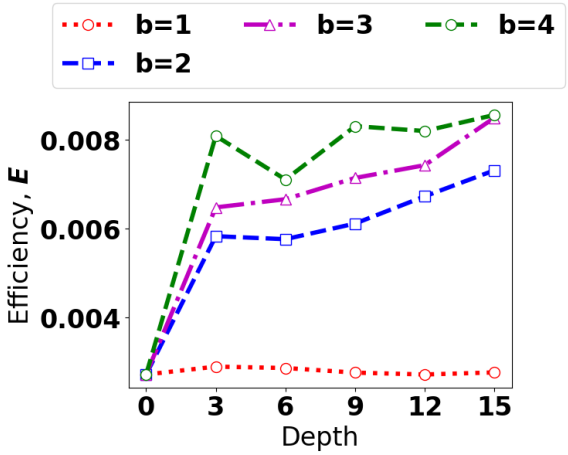


Figure 7: Efficiency E for various values of b and depth d in the Search and Rescue Domain for distance heuristic.

Depth and Running Time

Figure 10 shows how the acting time and the planning time changes with depth for the zero heuristic. We measured the acting time and planning time separately with the bolder lines denoting the acting time. We observe that acting time decreases and planning time increases with increase in depth d of RAEplan-LookAhead. This is expected because the planner needs more time to roll out upto greater depths and returns more efficient methods which in turn reduces the acting time. The changes with depth are more pronounced for $b = 4$ than $b = 1$ because RAEplan-LookAhead examines for method instances for higher values of b. We also observe that somewhere between $d = 9$ and $d = 12$, the planning time starts to dominate the acting time. This is interesting and may help one decide the ideal value of depth in their domain. The acting time and planning time for the distance heuristics is shown in Figure 11. The value of d may be chosen depending on the desired trade-off between efficiency and running time of RAEplan with respect to RAE. Note that

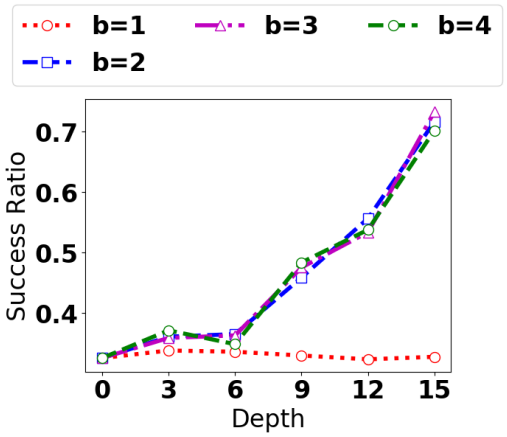


Figure 8: Success ratio (number of successful jobs/ total number of jobs) for various values of b and depth d in the Search and Rescue Domain for distance heuristic.

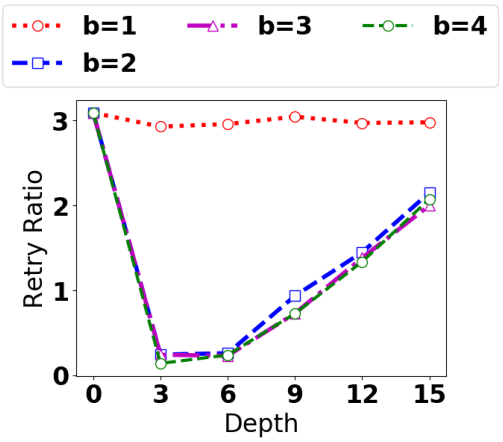


Figure 9: Retry ratio for various values of b and depth d in the Search and Rescue Domain for distance heuristic.

we also tried to observe patterns in the total time by taking an weighted sum of the acting and planning times. However, the results were not very meaningful in this domain.

Related Work

The approach to do planning in an operational model, RAE, and RAEplan, have been presented in (Patra et al. 2019). In this paper we discussed how the framework proposed in (Patra et al. 2019) can be used to easily interleave acting and planning and provided a novel experimental evaluation that shows the advantage of interleaving acting with planning. Beyond our AAAI work, to our knowledge, no previous approach has proposed the integration of planning and acting directly within the language of an operational model.

Our acting algorithm and operational models are based on the RAE algorithm (Ghallab, Nau, and Traverso 2016, Chapter 3), which in turn is based on PRS. If RAE and PRS need to choose among several eligible refinement methods for a given task or event, they make the choice without trying

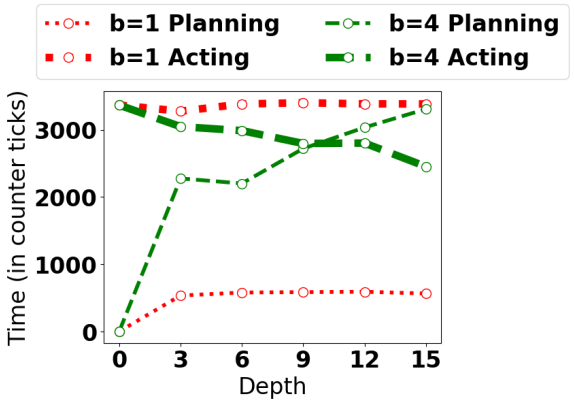


Figure 10: This figure shows that using the zero heuristic, the planning time (running time of RAEplan) increases with depth and the acting time (running time of RAE) decreases when it calls RAEplan with higher depth d. The time is measured in counter ticks. We do not show the plots for $b = 2$ and $b = 3$ here because they were similar to $b = 4$ and adding them made the figure more cluttered.

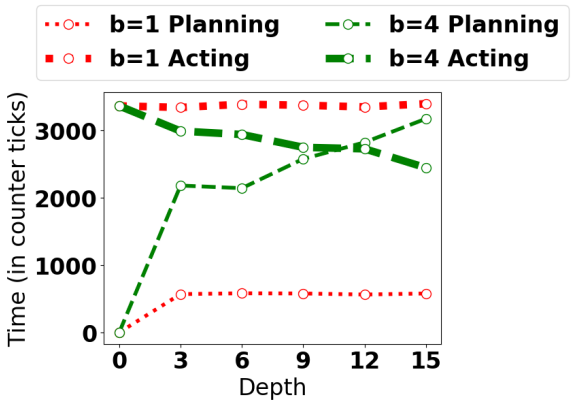


Figure 11: This figure shows that using the distance heuristic, the planning time (running time of RAEplan) increases with depth and the acting time (running time of RAE) decreases when it calls RAEplan with higher depth d. The time is measured in counter ticks.

to plan ahead. This approach has been extended with some planning capabilities in PropicePlan (Despouys and Ingrand 1999) and SeRPE (Ghallab, Nau, and Traverso 2016). Unlike our approach, those systems model commands as 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 RAE 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 have refinement capabilities and hierarchical models, e.g., (Verma

et al. 2005; Wang et al. 1991; Bohren et al. 2011). 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 do. Most of these systems do not reason about alternative refinements.

(Musliner et al. 2008; Goldman et al. 2016; Goldman 2009) propose a way to do online planning and acting, but their notion of "online" is different from ours. In (Musliner et al. 2008), the old plan is executed repeatedly in a loop while the planner synthesizes a new plan (which the authors say can take a large amount of time), and the new plan isn't installed until planning has been finished. In RAEplan, hierarchical task refinement is used to do the planning quickly, and RAE waits until RAEplan returns.

The Reactive Model-based Programming Language (RMPL) (Ingham, Ragno, and Williams 2001) is a comprehensive CSP-based approach for temporal planning and acting which combines a system model with a control model. 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 an extension of Simple Temporal Networks with symbolic constraints and decision nodes (Williams and Abramson 2001; Conrad, Shah, and Williams 2009). Planning consists in finding a path in the network that meets the constraints. RMPL has been extended with error recovery, temporal flexibility, and conditional execution based on the state of the world (Effinger, Williams, and Hofmann 2010). Probabilistic RMPL are introduced in (Santana and Williams 2014; Levine and Williams 2014) with the notions of weak and strong consistency, as well as uncertainty for contingent decisions taken by the environment or another agent. Our approach does not handle time; it focuses instead on hierarchical decomposition with Monte Carlo rollout and sampling.

Behavior trees (BT) (Colledanchise 2017; Colledanchise and Ögren 2017) can also respond reactively to contingent events that were not predicted. Planning synthesizes a BT that has a desired behavior. Building the tree refines the acting process by mapping the descriptive action model onto an operational model. Our approach is different since RAE provides the rich and general control constructs of a programming language and plans directly within the operational model, not by 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 and 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 in RAE. Moreover, these approaches integrate acting and planning without exploiting the hierarchical refinement approach described here.

Our methods are significantly different from those used in HTNs (Nau et al. 1999): to allow for the operational models needed for acting, we use rich control constructs rather than simple sequences of primitives. The hierarchical representation framework of (Bucchiarone et al. 2013) includes abstract actions to interleave acting and planning for composing web services—but it focuses on distributed processes, which are represented as state transition systems, not operational models. It does not allow for refinement methods.

A wide literature on MDP-based probabilistic planning and Monte Carlo tree search refers to simulated execution, e.g., (Feldman and Domshlak 2013; 2014; Kocsis and Szepesvári 2006; James, Konidaris, and Rosman 2017) and sampling outcomes of action models e.g., RFF (Teichteil-Königsbuch, Infantes, and Kuter 2008), FF-replan (Yoon, Fern, and Givan 2007) and hindsight optimization (Yoon et al. 2008). The main conceptual and practical difference with our work is that these approaches use descriptive models, i.e., abstract actions on finite MDPs. Although most of the papers refer to doing the planning online, they do 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.

Finally, there has been a lot of work in robotics to integrate planning and execution. They propose various techniques and strategies to handle the inconsistency issues that arise when execution and planning are done with different models. (Lallement, De Silva, and Alami 2014) shows how HTN planning can be used in robotics. (Garrett, Lozano-Perez, and Kaelbling 2018a) and (Garrett, Lozano-Pérez, and Kaelbling 2018b) integrates task and motion planning for robotics. (Coste-Maniere et al. 2017) integrates planning and execution for surgical planning algorithms used by surgical robots for laporoscopic and other minimally invasive surgery.

Conclusions and Future Work

In this paper, we discussed different ways to interleave acting and planning using operational models. Our actor is RAE and our planner is RAEplan-LookAhead. We came up with simple domain dependent heuristics in a simulated domain, called Search and Rescue and showed that performance improves with depth but the cost is that it also takes more time. Depending on the domain, the heuristics available and the performance requirements, the set of experiments done in this paper can help one identify the sweet spot.

In future, we plan to interleave acting and planning in some other domains and observe the changes in the performance of RAE and RAEplan-LookAhead. We also plan to do experiments using the lazy and concurrent strategies discussed in this paper.

References

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

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

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

Coste-Maniere, E.; Adhami, L.; Boissonnat, J.-D.; Carpentier, A.; and Guthart, G. S. 2017. Methods and apparatus for surgical planning. US Patent App. 15/397,498.

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

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

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

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

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

Garrett, C. R.; Lozano-Perez, T.; and Kaelbling, L. P. 2018a. Ffrob: Leveraging symbolic planning for efficient task and motion planning. *The International Journal of Robotics Research* 37(1):104–136.

Garrett, C. R.; Lozano-Pérez, T.; and Kaelbling, L. P. 2018b. Stripstream: Integrating symbolic planners and blackbox samplers. *arXiv preprint arXiv:1802.08705*.

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

Goldman, R. P.; Bryce, D.; Pelican, M. J.; Musliner, D. J.; and Bae, K. 2016. A hybrid architecture for correct-by-construction hybrid planning and control. In *NASA Formal Methods Symposium*, 388–394. Springer.

Goldman, R. P. 2009. A semantics for htn methods. In *ICAPS*.

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

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

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

Lallement, R.; De Silva, L.; and Alami, R. 2014. Hatp: An htn planner for robotics. *arXiv preprint arXiv:1405.5345*.

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

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.

Musliner, D. J.; Pelican, M. J.; Goldman, R. P.; Krebsbach, K. D.; and Durfee, E. H. 2008. The evolution of circa, a theory-based ai architecture with real-time performance guarantees. In *AAAI Spring Symposium: Emotion, Personality, and Social Behavior*, volume 1205.

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

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

Patra, S.; Gallab, M.; Nau, D.; and Traverso, P. 2018. Using operational models to integrate acting and planning. In *IntEx: ICAPS 2018 Workshop on Integrated Planning, Acting and Execution*.

Patra, S.; Gallab, M.; Nau, D.; and Traverso, P. 2019. Acting and planning using operational models. In *AAAI*. AAAI Press.

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

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

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

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*.

Vaquero, T.; Roberts, M.; Bernardini, S.; Niemueller, T.; and Fratini, S., eds. 2018. *Proceedings of the 2nd Workshop on Integrated Planning, Acting, and Execution*, ICAPS 2018 Workshop.

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

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

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

Executing Multi-Goal Mission Plans for Coordinated Mobile Robots

Marlyse Reeves, Enrique Fernández-González, Brian Williams
MIT, CSAIL
Cambridge, MA 02139
{mreeves,efernan,williams}@csail.mit.edu

Abstract

This paper presents a centralized executive for robotic mission plans with multiple temporally extended goals and coordinated agents. Common approaches to online motion planning and execution execute sequential goals independently, and do not consider the full plan when planning for the next goal. This often results in suboptimal or infeasible plans. Some hybrid and temporal planners are capable of generating motion plans for multiple vehicles with multiple goals over long horizons. However, the dynamics considered are often too simple to be executed by real systems. We present an executive planner that plans local trajectories using sufficiently accurate dynamics while considering the rest of the global plan constraints in the far future. We achieve this by repeatedly solving a single, multi-fidelity optimization problem, where we combine higher fidelity discrete-time dynamics to generate the local trajectory and lower fidelity continuous-time dynamics to capture the full remaining plan, guiding the local trajectory. We evaluate our multi-goal executive planner against a naive, myopic executive and demonstrate the scalability on a set of expressive real-world scenarios.

1 Introduction

Autonomous mobile systems are being tasked to execute increasingly complex missions in a variety of domains. For example, NASA plans to send a suite of remote agents to Saturn's moon Europa to autonomously explore the icy surface and oceans for long durations without human intervention. Robotic scenarios like this often involve coordination of multiple heterogeneous vehicles subject to expressive constraints including temporal deadlines, limited resources, and coupling between vehicles. Mission plans generated for these scenarios require agents to achieve multiple goals over long time spans. Local execution may affect the feasibility of goals later in the plan. Similarly, goals in the future may impose constraints on the immediate behavior of the system.

To successfully execute these challenging missions, an executive planner must 1) avoid being myopic by considering the constraints of the full plan to ensure agents can achieve all the mission goals, 2) generate dynamically feasible trajectories that respect the complex constraints of the mission, and 3) reason in real time to adapt to changes in the environment and uncertainty in agent execution.

Traditionally, to ensure sufficiently fast planning, executives only reason about the next goal. Model predictive control (MPC) is a widely used approach for this execution strategy. This approach solves a series of local control problems iteratively until the goal is reached, allowing for plan adaptation in response to disturbances. Typically, MPC methods use spatial heuristics to guide the local trajectory (Mettler, Dadkhah, and Kong 2010), (Bellingham, Kuwata, and How 2003), (Kuwata and How 2004). Again, these approaches only consider a single goal and are often limited to single agent problems with simple linear constraints.

Other work addresses efficient long horizon execution planning with finer resolution as the plan is executed (Jain and Tsiotras 2009). In these multi-fidelity approaches, only simple constraints for a single vehicle are considered. Hierarchical approaches use a lower level trajectory planner to refine solutions generated by a high level planner, speeding up the planning process to allow for larger and more complex problems. One such approach uses a constrained Markov decision process (CMDP) as a mission planner integrated with a probabilistic roadmap (PRM) for more fine-grained control to handle scenarios in complex cityscapes (Ding et al. 2014). Another approach solves a graph search problem on a discretized map using a continuous trajectory planner to compute edge costs and ensure dynamic feasibility (Cowlagi and Tsiotras 2012). However, during execution, these approaches only use the next unsatisfied goal to guide the local trajectory.

Work has been done to consider multiple goals during planning. One approach models the mission activities as a directed graph encoded with spatial and temporal constraints (Bhattacharya, Likhachev, and Kumar 2010). Although this approach can handle complex problems with multiple coordinated vehicles, the graph search is too slow for online execution. Other approaches use a similar graph-like formulation encoded as a receding horizon optimization problem (Léauté and Williams 2005), (Hofmann and Williams 2017). These approaches demonstrate robust execution of complex, multi-goal mission plans and serve as the foundation of our work. We give a more efficient optimization encoding with a

more informative heuristic guidance which allows for real-time execution of richer mission plans.

Hybrid planners are devised to handle coordinated agents over long horizons with multiple goals (Fernández-González, Williams, and Karpas 2018). However, these focus on the generative activity planning side, and only consider simple first-order constant velocity dynamics that are not suitable for direct execution. The temporal logic community has also frequently addressed the problem of planning multi-goal, long duration missions for multiple vehicles (Wurm et al. 2013) (Ulusoy et al. 2012) (Schillinger, Bürger, and Dimarogonas 2018). These planners efficiently assign and schedule tasks to many agents however the plans consist of a series of waypoints or graph nodes which cannot be directly executed by a real robot. Our executive planner does not do task allocation or activity planning but instead could support these planners by providing centralized, multi-vehicle trajectory planning and execution for long horizon mission plans.

In this paper, we introduce a centralized executive planner capable of executing multi-goal, long duration missions involving multiple coordinated vehicles. Leveraging a multi-fidelity approach, our executive plans trajectories using a detailed discrete-time formulation for a limited horizon while simultaneously reasoning over the entirety of the remaining plan using a lower fidelity continuous time formulation. The key innovation of our approach is how we combine this local (discrete time) and global (continuous time) planning into a single optimization problem such that the local trajectory is guided by the constraints and objectives of the global mission plan. Our executive planner solves this optimization problem iteratively, in real-time, using a receding horizon approach which allows us to react to disturbances discovered during execution. In our discussion of this work, we assume that mission plans have been generated by experts or by a generative hybrid planner similar to ScottyActivity (Fernández-González, Williams, and Karpas 2018). We empirically evaluate our executive planner on a set of real-world multi-agent scenarios in simulation and show that it can execute a richer set of mission scenarios when compared to myopic online executives.

2 Motivating Scenario

Our team at MIT will participate in a robotic mission in Santorini, Greece to study marine life around an underwater volcano. In this work, we will use an idealization of this mission to illustrate the capabilities of our executive planner. The scenario involves an Autonomous Underwater Vehicle (AUV) and a ship. The AUV uses a thruster powered by a battery to propel itself through the water.

Consider an example mission plan for this scenario. Initially, the ship is transporting the AUV to the area of scientific interest. First, the AUV must take images of the sea floor in region A. Then, the AUV must visit region B to take samples of the water column. After all the data has been gathered, the ship must recover the AUV within 30 minutes. Finally, the ship, with the AUV on board, must end in the destination region. For safety reasons, the AUV must stay within communication range of the ship at all times. Additionally, the AUV must avoid consuming the entire battery supply during the mission. We assume all obstacles are below the ocean surface and therefore only the AUV is required to avoid them. The objective of this mission is to minimize a linear combination of the total mission time and the total distance traveled by the ship. Figure 1a shows an example plan satisfying the mission goals in 2D.

This mission plan is flexible with implicit goals; the planner must decide, for example, where and when the AUV and ship should rendezvous. Also, notice that the trajectories executed in the near term affect the rest of the plan. For instance, if the ship deploys the AUV close to region A, it will have more battery power to use for the rest of the mission. Additionally, the constraints later in the plan affect early execution. For example, the AUV must achieve all its science goals fast enough to rendezvous with the ship before the deadline. Moreover, the AUV and the ship are tightly coupled throughout the mission due to the communication range constraint.

3 Problem Statement

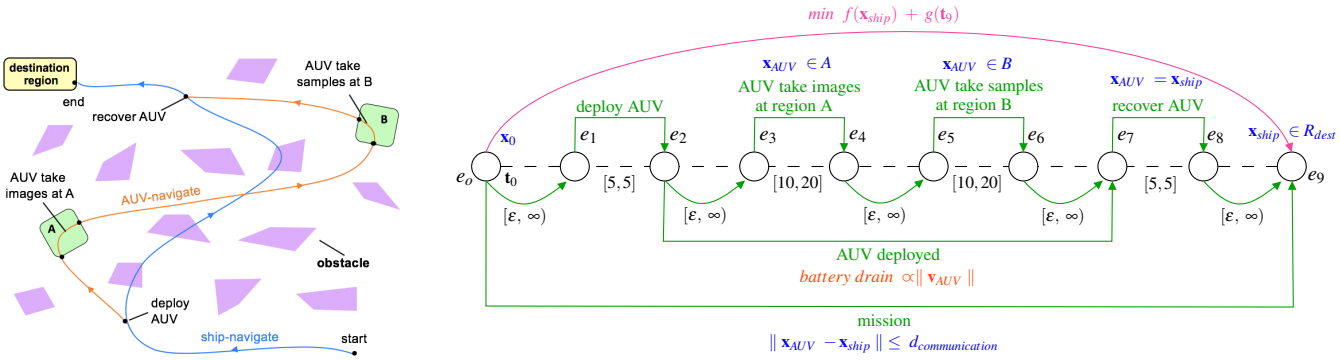
Given a representation of the mission plan specifying vehicle dynamics, multiple goals over time, and mission constraints, our executive planner finds a finite horizon control sequence. This control sequence may satisfy some of the mission goals that are achievable during this finite horizon and will ensure that the remaining goals are achievable under relaxed dynamics. Finally, this control sequence is guided by all the remaining goals of the full mission plan. In order to the complete the mission, we iteratively plan and execute control sequences using a receding horizon formulation until all the goals are satisfied.

3.1 Definition of a Qualitative State Plan

A complex robotic mission plan can be represented as a *qualitative state plan* (QSP) (Léauté and Williams 2005). This graph-like representation uses nodes to represent distinct points in time, or *events*. Edges, or *episodes*, capture state constraints, temporal constraints, or vehicle dynamics that are active between a pair of events.

More formally, a $QSP = \langle X, U, E, E_P, O \rangle$ describes the state, temporal, and dynamic constraints that need to be satisfied at different points in the plan where:

- X is a set of state variables for all agents.
- U is a set of control variables for all egents.
- E is an ordered list of events with each event e given by $\langle t_e, S_e \rangle$ where:
  - t_e is a real-valued variable representing the execution time of the event.
  - S_e is the set of state constraints on the values of the state variables at that event.
- E_P is the set of episodes and each episode ep is given by $\langle e_S, e_E, d_l, d_u, S_{ep}, D_{ep} \rangle$ where:
  - e_S and e_E are the starting and ending events of the episode.



(a) Ocean exploration motivating scenario plan. The ship and AUV trajectories are shown in blue and orange respectively.

(b) Qualitative state plan for the ocean exploration motivating scenario. The plan consists of activities that must be executed in a certain order. Activities have associated constraints that restrict the state of the vehicles, describe the dynamical behavior, and impose temporal relations between other activities.

Figure 1: Ocean Exploration Motivating Scenario

  - d_l and d_u are the lower and upper bounds on the duration of the episode.
  - S_{ep} is the set of state constraints that must be satisfied for the duration of the episode.
  - D_{ep} is the set of dynamic equations that relate control inputs to state constraints, describing how state variables change over the duration of the episode.
- O is an objective to be minimized over the entire mission as a function of state and control variables.

Additionally, we assume the the events in the qualitative state plan are *totally ordered* such that $t_1 < t_2 < \cdots < t_m$ where m is the total number of events.

The qualitative state plan for our motivating scenario is shown in Figure 1b. The events are shown as black circles. For example, event e_3 is the moment in time when the AUV starts taking images of region A. The episodes are shown in green. A subset of the state and dynamic constraints are shown in blue and orange respectively. The episode between events e_5 and e_6 represents the AUV taking samples in region B. This activity can take between 10 and 20 minutes and requires that the AUV be inside region B for the duration of the episode. The AUV is deployed and navigating underwater between events e_2 and e_7 during which time its battery drains as a function of its velocity. The objective over the entire mission is shown in the QSP in pink. Finally, the $[\varepsilon, \infty)$ temporal constraints between some events enforce the total ordering of events (i.e. e_1 must happen at least ε minutes after e_0 where ε is some small constant).

3.2 Multi-goal Execution Problem

In order to successfully execute the mission described by the QSP, we iteratively find finite control trajectories for a receding horizon. We denote *multi-goal execution problem* as the problem of finding such local finite trajectory for the next planning horizon while considering the goals of the full plan. Our multi-goal execution problem is given by the 6-tuple, $\langle QSP, \boldsymbol{x}_o, e_{next}, n, \Delta t, env \rangle$ where:

- QSP is a qualitative state plan.
- $\boldsymbol{x}_o$ is the current state of all agents.
- e_{next} is the next event in the QSP that is yet unsatisfied.
- n is the number of discrete time steps.
- Δt is the sample time between discrete time steps.
- env is a representation of the obstacles in the environment.

The problem consists of building and solving a trajectory optimization problem to generate a finite control trajectory $\langle \boldsymbol{u}_0, \ldots, \boldsymbol{u}_{n-1} \rangle$ and the corresponding state trajectory $\langle \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \rangle$ for each discrete time step $t_0, \ldots, t_{n-1}$. We define the *planning horizon*, T_P, to be the total time of this finite control trajectory.

$$T_P = t_n = t_0 + (n-1)\Delta t$$

Note that in general, $T_P \ll T_M$, where T_M is the total time of the entire executed mission. Additionally, we let $T_h < T_P$ where T_h is the *execution horizon*, or the length of the control trajectory actually executed by the agents before replanning. A *valid* finite control trajectory and corresponding state trajectory satisfies all constraints and dynamics captured by the corresponding portion of the QSP and avoids obstacles in the environment. This finite trajectory might satisfy one or more of the remaining events in the QSP if those events are achievable within the execution horizon.

4 Approach

Our executive planner hinges on a "multi-fidelity" approach; that is, *jointly* reasoning with a high fidelity model just before execution while reasoning with a lower fidelity model for the remaining mission plan. More specifically, we use a discrete time formulation to generate a finite horizon trajectory. This formulation allows us to reason about executable dynamics and local hazards for a limited time to avoid intractability. Conversely, we use a continuous time formulation to capture the remaining mission plan. While this formulation is lower resolution, it allows us to efficiently reason

over long and complex plans. We unify these discrete time and continuous time formulations into a single optimization problem which is solved iteratively, in real time, until all the mission goals have been satisfied. Most importantly, this integration ensures that the local finite trajectory respects and is informed by the constraints and goals of the overall mission plan.

In the following section, we first review how our approach leverages existing discrete time and continuous time trajectory optimization formulations, then discuss our main innovation which is the integration of these two formulations for multi-goal execution guided by the full plan.

4.1 Generating Finite Control Trajectories

To enable successful missions, we must generate detailed and accurate control trajectories that agents can execute in the real world. In addition to being dynamically feasible, these finite control trajectories must respect the constraints of the QSP and avoid local hazards such as nearby obstacles or agents. We use a discrete time trajectory optimization formulation to generate these high-fidelity trajectories. More specifically, we use a fixed horizon of n discrete time steps where Δt is the sample time.

Our discrete time formulation, derived from generic trajectory optimization methods such as the one presented in (Schouwenaars et al. 2001), can be written as

$$\begin{aligned}
\min_{\boldsymbol{u}_i, \boldsymbol{x}_n} \quad & \sum_{i=0}^{n-1} \boldsymbol{r}|\boldsymbol{u}_i| + f(\boldsymbol{x}_n) && \text{(1a)}\\
\text{subject to} \quad & \boldsymbol{x}_{i+1} = \boldsymbol{A}\boldsymbol{x}_i + \boldsymbol{B}\boldsymbol{u}_i, && \text{(1b)}\\
& g_k(\boldsymbol{x}_i) \leq 0, && \text{(1c)}\\
& \boldsymbol{u}_{\min} \leq \boldsymbol{u}_i \leq \boldsymbol{u}_{\max}, && \text{(1d)}\\
& i = 0, \ldots, n-1
\end{aligned}$$

where $\boldsymbol{x}$ and $\boldsymbol{u}$ are the state and control vectors for all vehicles. $\boldsymbol{A}$ and $\boldsymbol{B}$ are matrices that define second-order linear state space dynamics models for all vehicles. In addition to dynamics, finite control trajectories are subject to convex state constraints (Eq. 1c) and actuation limits (Eq. 1d). Agents are assumed to start at a known initial state. The cost function given in Eq. 1a uses a weight one norm of the control, where r is a non-negative weight matrix, that represents total control effort. $f(\boldsymbol{x}_n)$ is some terminal cost on the final state, usually the approximate cost or distance to the goal. In our formulation, the state constraints and dynamics are derived from the QSP which will be discussed in section 4.3.

In our motivating scenario, the state vector contains the position and velocities for both vehicles as well as the battery level for the AUV. The control vector contains the accelerations for both vehicles.

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{x}_{auv} \\ \boldsymbol{x}_{ship} \end{bmatrix}, \quad \boldsymbol{x}_{auv} = \begin{bmatrix} x_{auv} \\ y_{auv} \\ v_{x,auv} \\ v_{y,auv} \\ b_{auv} \end{bmatrix}, \quad \boldsymbol{x}_{ship} = \begin{bmatrix} x_{ship} \\ y_{ship} \\ v_{x,ship} \\ v_{y,ship} \end{bmatrix} \tag{2}$$

$$\boldsymbol{u} = \begin{bmatrix} \boldsymbol{u}_{auv} \\ \boldsymbol{u}_{ship} \end{bmatrix}, \quad \boldsymbol{u}_{auv} = \begin{bmatrix} a_{x,auv} \\ a_{y,auv} \end{bmatrix}, \quad \boldsymbol{u}_{ship} = \begin{bmatrix} a_{x,ship} \\ a_{y,ship} \end{bmatrix} \tag{3}$$

Obstacles are represented by convex polygons $P := \{x|\boldsymbol{A}\boldsymbol{x}_p < \boldsymbol{b}\}$ where $\boldsymbol{x}_p$ is a positional state vector. To avoid collision with polygon P, the following constraint must be satisfied

$$\boldsymbol{a}_i^T \boldsymbol{x}_p \geq b_i - M d_i, \quad i = 1, \ldots, D \tag{4}$$

$$\sum_{i=1}^{D} d_i \leq D - 1 \tag{5}$$

where $\boldsymbol{a}_i$ is a row in matrix A, $d_i \in \{0,1\}$ are binary decision variables, M is a large constant, and D is the number of sides of the polygon. A similar method can be used to model collision avoidance between vehicles (Richards et al. 2002).

4.2 Reasoning Over Long Horizons

In addition to generating local finite control trajectories, we would like to be able to reason about the full mission plan. A naive approach would be to extend the discrete time approach described in the previous section to the full plan horizon. While this would give an optimal control trajectory for the full plan that satisfies all the state and dynamic constraints, increasing the number of discrete time steps increases the number of variables and constraints in the problem. For the mission durations that we are interested in, the problem becomes intractable. Instead, our approach avoids fine-grained time discretization by using a continuous time optimization formulation to reason about the full plan.

We leverage a class of convex optimization programs called second order cone programs (SOCP) that are fast to solve and allow for constraints that commonly arise in multi-vehicle mission plans. More specifically, SOCPs allow us to express inequality constraints on the norm and squared norm of vectors of variables. For example, we can model the AUV's battery decrease as function of the magnitude of the velocity vector and enforce a maximum distance constraint between the AUV and ship. In this work, we use the convex model presented in (Fernández-González, Williams, and Karpas 2018).

We encode the QSP, which represents the entire mission plan, as a continuous time SOCP optimization problem. To aid in our discussion, we define a *segment* to be the period of time between two successive events in the QSP. Recall that events are totally ordered in our definition of QSPs. Also note that segments are different from episodes which can span multiple events (and therefore multiple segments).

At each event j, we define a vector of state variables $\boldsymbol{q}_j$ and for each segment k we define a vector of control variables $\boldsymbol{c}_k$. Additionally, we add a variable t_j to represent the

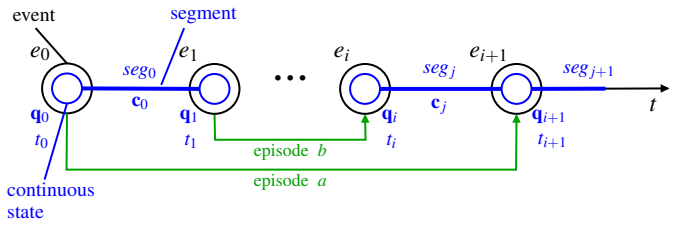


Figure 2: Continuous time formulation overlaid on a QSP. Events and segments and their associated variables are depicted in blue.

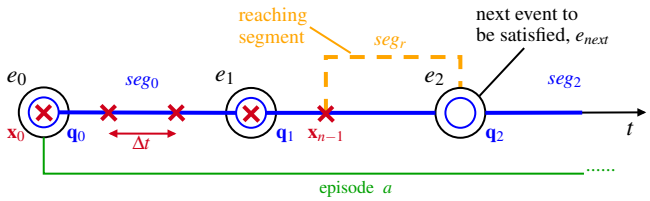


Figure 3: Integrated discrete and continuous formulations for a single planning horizon overlaid on a QSP. Local trajectory with a fixed time step Δt is represented by red x's. The planning horizon $T_p = 4 \cdot \Delta t$. The reaching segment is depicted in orange.

time at each event j. Figure 2 shows the continuous time formulation overlaid on the QSP.

For reasoning over the full plan, we restrict the dynamics to first-order velocity control and constrain the control variables to take constant values between events. For our motivating example, the state and control vectors in the full plan formulation are

$$\boldsymbol{q} = \begin{bmatrix} \boldsymbol{q}_{auv} \\ \boldsymbol{q}_{ship} \end{bmatrix}, \quad \boldsymbol{q}_{auv} = \begin{bmatrix} x_{auv} \\ y_{auv} \\ b_{auv} \end{bmatrix}, \quad \boldsymbol{q}_{ship} = \begin{bmatrix} x_{ship} \\ y_{ship} \end{bmatrix} \tag{6}$$

$$\boldsymbol{c} = \begin{bmatrix} \boldsymbol{c}_{auv} \\ \boldsymbol{c}_{ship} \end{bmatrix}, \quad \boldsymbol{c}_{auv} = \begin{bmatrix} v_{x,auv} \\ v_{y,auv} \end{bmatrix}, \quad \boldsymbol{c}_{ship} = \begin{bmatrix} v_{x,ship} \\ v_{y,ship} \end{bmatrix} \tag{7}$$

Our state transition equations becomes

$$\boldsymbol{q}_{j+1} = \boldsymbol{q}_j + \boldsymbol{c}_j \cdot (t_{j+1} - t_j) \tag{8}$$

Notice that this equation is non-linear and non-convex and cannot be directly encoded in a convex program. (Fernández-González, Williams, and Karpas 2018) introduces a proxy variable $\boldsymbol{c}_{j\Delta t_j} = \boldsymbol{c}_j \cdot (t_{j+1} - t_j)$ and reasons over this variable in the optimization. We can see that substituting $\boldsymbol{c}_{j\Delta t_j}$ into Eq. 8 makes the state evolution linear. More details on this proxy decision variable and how it impacts the constraints can be found in (Fernández-González, Williams, and Karpas 2018).

Given that the variables in our formulation correspond directly to events and episodes, we can directly apply constraints to these variables from the QSP. More specifically, we add state constraints on the variables in $\boldsymbol{q}_j$ from event j as well as from any episode spanning that event. Similarly, the control variables in $\boldsymbol{c}_k$ become constrained by the dynamic constraints of any episodes segment k is contained in. Finally, the duration bounds on episodes enforce constraints on the time variables t_j.

The solution to our continuous time formulation over the full plan is a piece-wise linear state trajectory that ignores obstacles. Note that this "relaxed" solution is always be optimistic with respect to the discrete time formulation and therefore serves as an admissible heuristic for the local finite control trajectory. By leveraging convex optimization, the continuous time solution can be efficiently computed for long duration mission plans with many goals online.

4.3 Multi-goal Execution

To successfully execute multi-vehicle mission plans, our executive planner must ensure that the local trajectory obeys the constraints of the QSP and that all events in the QSP are executed successfully and in the proper order. Additionally, we'd like the local finite control trajectory to be guided by the full plan. In the following section, we describe how our approach to integrating the discrete time and continuous time formulations into a single optimization problem meets these needs. More specifically, our trajectory optimization problem is a mixed integer second-order cone program (MISOCP). For reference, Fig. 3 shows both the discrete and continuous time formulations for a single planning horizon overlaid on the QSP.

The local state and control trajectory must obey the constraints of the QSP. However, as shown in Fig. 3, states in the local trajectory do not necessarily fall between the same two events and thus different constraints may apply at different points in the trajectory. For example, in our motivating scenario, the AUV only needs to be in region A between events e_3 and e_4. We use binary indicator variables to choose which constraints are active at each discrete time step.

Assume that our QSP has m events (and therefore $m-1$ segments). Additionally, there are n discrete time states where $n = T_p/\Delta t$. We introduce the following *segment indicator* binary variables

$$z_{ik} = \begin{cases} 1 & \text{if } \boldsymbol{x}_i \text{ in segment } k, \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

$$i = 0, \ldots, n-1, \quad k = 0, \ldots, m-2$$

$$\sum_{k=0}^{m-1} z_{ik} = 1 \tag{10}$$

where $z_{ik} \in \{0,1\}$ and $\boldsymbol{x}_i$ are discrete time states. For example, from Fig. 3, we see that $z_{0,0} = z_{1,0} = z_{2,0} = 1$ because states $\boldsymbol{x}_{0\ldots2}$ fall in segment 0. We also impose the constraint (Eq. 10) that each state on the local trajectory must be assigned to exactly one segment in the QSP. If a local state is assigned to segment k, it inherits the state and dynamic constraints from all the episodes that span that segment k.

Our formulation also needs to ensure that all events in the QSP are executed in the correct order while obeying their state constraints. Because events can be executed at any time (subject to temporal constraints) where as the local trajectory is sampled at fixed time steps, our executive planner must decide which events, if any, will be satisfied during each planning horizon.

We add the following *state indicator* variables to the model to allow the planner to make that choice.

$$w_{ij} = \begin{cases} 1 & \text{if event } j \text{ is satisfied by } \boldsymbol{x}_i, \\ 0 & \text{otherwise} \end{cases} \tag{11}$$

$$i = 0,\ldots,n-1, \quad j = 0,\ldots,m-1$$

$$\sum_{i=0}^{n-1} w_{ij} \leq 1 \tag{12}$$

where $w_{ij} \in \{0,1\}$. For example, in Fig. 3, $w_{n-2,1} = 1$ since event e_1 is satisfied by state x_{n-2}. We also impose the constraint (Eq. 12) that each event can be satisfied by at most one state in the local trajectory. We also use this constraint as a stopping condition for our executive planner. The last planning horizon is when the final event in the QSP is satisfied and scheduled to be executed within the execution horizon, T_h.

Because our local control trajectory has fixed time steps, we cannot impose arbitrary temporal constraints directly on the discrete time variables. However, our approach uses a combination of the state and segment indicator variables to ensure that events are executed sequentially in the correct order. More explicitly we add the following constraints to our formulation

$$w_{ij} = 1 \iff \sum_{l=0}^{i} w_{l(j-1)} = 1 \tag{13}$$

$$z_{ik} = 1 \iff \sum_{l=0}^{i} w_{lk} = 1 \tag{14}$$

Intuitively, Eq. 13 states that for discrete time state $\boldsymbol{x}_i$ to satisfy event j in this planning horizon, event $j-1$ must have been satisfied by a previous state in the local control trajectory in this planning horizon (unless $j = 0$). Eq. 14 states that a discrete time state $\boldsymbol{x}_i$ can only be assigned to segment k if event k was satisfied at or before $\boldsymbol{x}_i$ or the planning horizon started within segment k.

Recall that when reasoning over the full plan, we assigned a state vector q_j to each event j in the QSP. If x_i satisfies event j in the local trajectory then it must be consistent with the state in the global "relaxed" trajectory, inheriting its state constraints.

$$w_{ij} = 1 \iff \begin{cases} \boldsymbol{x}_i = \boldsymbol{q}_j \\ T_i = t_j \end{cases} \tag{15}$$

$$i = 0,\ldots,n-1, \quad j = 0,\ldots,m-1$$

where $T_i = t_0 + i \cdot \Delta t$ is planned execution time of $\boldsymbol{x}_i$. Note that the state variable vectors differ between the two models. $\boldsymbol{x}$ contained position, velocity, and battery level while $\boldsymbol{q}$ only contained position and battery level. Eq. 15 only applies to the state variables present in both $\boldsymbol{x}$ and $\boldsymbol{q}$ state vectors.

Our executive planner not only ensures consistency with the QSP but also forces the local trajectory to be guided by the global plan. For guidance, the executive planner must know which event in the QSP is next to be executed *after* the current planning horizon. We refer to this event as the "chased event", e_{chased}. Some event $e_j = e_{chased}$ if all

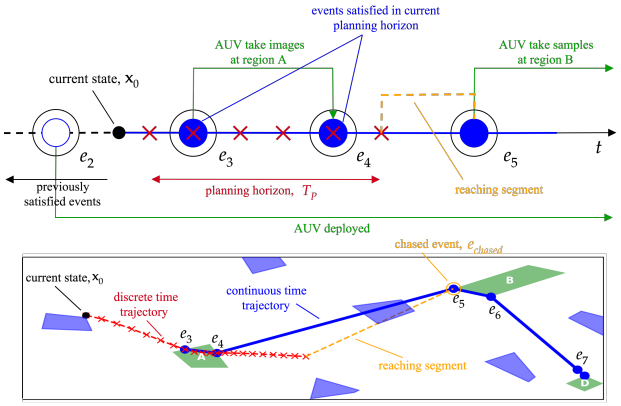


Figure 4: (top) Integrated discrete and continuous formulation for a single planning horizon overlaid on the motivating example QSP. Note that the discrete states are not shown to scale. (bottom) Integrated discrete and continuous time trajectories for single planning horizon during execution plotted in mission environment for the motivating scenario. The ship's trajectory is omitted for simplicity.

the events before event j have already been executed or are scheduled to be executed in the current planning horizon. We use the segment indicators to identify the chased event

$$z_{n(j-1)} = 1 \iff e_j = e_{chased} \tag{16}$$

$$e_j = e_{chased} \iff \boldsymbol{q}_{chased} = \boldsymbol{q}_j \tag{17}$$

Intuitively, if the last local trajectory state $\boldsymbol{x}_n$ is going to be executed in segment $j-1$, then the chased event is e_j and the global state associated with e_j becomes the chased state.

We introduce a new auxiliary segment called the *reaching segment*, which is the period of time between the last state in the local trajectory and the global state $\boldsymbol{q}_{chased}$. We also define a new constant control variable vector for this segment, $\boldsymbol{c}_r$. The state evolution for the reaching segment becomes

$$\boldsymbol{q}_{chased} = \boldsymbol{x}_n + \boldsymbol{c}_r \cdot (t_{chased} - T_n) \tag{18}$$

where $T_n = t_0 + n \cdot \Delta t$ is planned execution time of $\boldsymbol{x}_n$. The reaching segment is show in orange in Fig. 3.

Eq. 18 imposes a constraint on the global model that affects the overall optimization cost. For example, assume the objective is to minimize the total time of the full plan. The above constraint implies that the fastest you can get to the next event after the planning horizon is from the last point in the local trajectory going at full speed in a straight line. This may take longer than the original global plan. The optimization will try to minimize the reaching segment and direct the local trajectory towards the next event. Additionally, since the next event constrains the future events in the global plan, the local trajectory execution also impacts states in the far future. Fig. 4 shows the local and global trajectories for the AUV in a planning horizon during the execution of our motivating scenario. The last point in the local trajectory is constraining the next event in the QSP through the reaching segment shown in dotted orange.

	Our Executive		Naive Executive	
Scenario	OBJ	t	OBJ	t
One AUV	46.53	0.22	34.77	0.19
AUV + Ship	17.54	0.35	22.97	0.20
Hazard Reg.	11.02	0.31	19.44	0.10
Motiv. Scen.	19.28	0.47	-	-

Table 1: Plan quality comparison. **OBJ**: Mean objective value; **t**: Mean optimization time for single planning horizon in seconds. Example solutions for these scenarios are shown in Fig. 5

.

5 Experimental Results

Our executive planner is implemented in Python and uses the GUROBI 8.0.1 solver to solve the MISCOP optimization problems. We compare our executive planner to a naive, myopic executive and evaluate its performance and scalability on a variety of scenarios.

For comparison, we implement a "naive" executive that only considers the next event in the QSP when planning local trajectories. More specifically, this naive executive uses the same discrete time formulation presented in Sec. 4.1 with the next unsatisfied event encoded as the goal. Using a receding horizon approach, the naive executive plans finite trajectories iteratively until the event is satisfied. Then, it takes the next unsatisfied event in the QSP as input and repeats the process until all the events are satisfied. Importantly, the naive approach is myopic; it only considers the next event during planning instead of the entire remaining mission plan. The naive executive minimizes a linear combination of the local control effort and the time to the next event while our executive planner minimizes a linear combination of the local control effort and the time of the last event in the QSP. To compare the *plan quality,* or value of the objective function, of our executive planner and the naive executive, we use 4 illustrative scenarios. For each scenario, we ran 60 trials with randomized initial conditions. A summary of the results can be seen in Table 1 and we will briefly touch on each scenario.

First, we consider a simple single vehicle scenario. In this first scenario, a single AUV visits regions while avoiding obstacles. The naive executive is a greedy approach so for this and other simple scenarios it gives better quality plans than our executive.

Next, we consider scenarios where early decisions impact the quality of the overall plan. In the second scenario, an AUV is deployed by a ship to visit region A then region B (Fig. 5a). The AUV does not need to conserve battery and there is no communication constraint between the AUV and the ship however the ship must still recover the AUV and travel to the destination region before the end of the mission. The naive executive does not know that the ship and the AUV must be at the same location later in the plan. As a result, it makes no effort early on to move the ship in the same direction as the AUV and the AUV must double back to rendezvous with the ship, following a suboptimal route. On the other hand, our executive planner uses the full QSP to guide the ship's local trajectory towards the destination where it meets and recovers the AUV. The third scenario is the same as the second execept now there is a hazard cutting through region A (Fig. 5b). The naive executive greedily steers the AUV to the closest point in region A, causing the trajectory to region B to be suboptimal. Our executive planner steers the AUV to a location in region A farther from the AUV's initial position but creating a more efficient trajectory through region B overall.

Finally, we consider our motivating scenario. Recall from Sec. 2, that there is a communication range constraint between the AUV and the ship, a temporal constraint over the entire mission, and a limited battery supply for the AUV. For this scenario, the naive executive, unaware of the rest of the plan, generates a greedy trajectory to region A, causing the AUV to run out of battery. In contrast, our executive planner is able to generate a high quality plan that obeys all the constraints. As Fig. 5c shows, the ship recovers the AUV before it runs out of battery and then navigates, with the AUV on-board, to the destination.

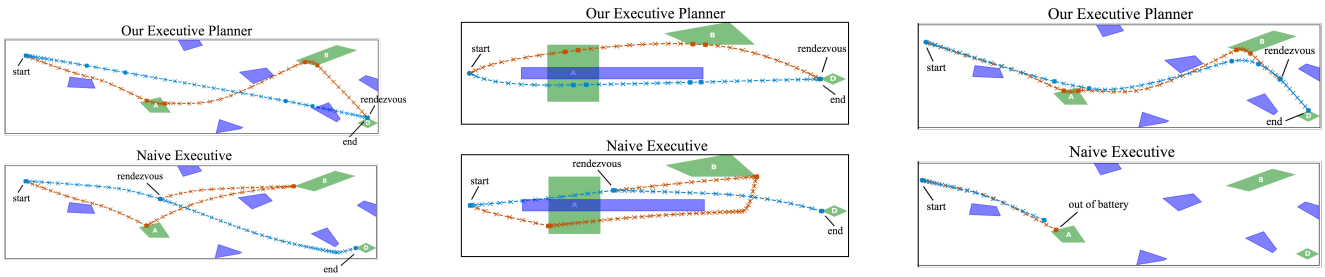
To evaluate the scalability of our executive planner, we tested four different ocean exploration scenarios on increasingly complex mission plans. Since the size of the optimization problem decreases as the mission is executed and events are satisfied, we only consider the optimization time to solve the first MISOCP at the beginning of each scenario Figure 6 shows the results that were obtained on an Intel i9 7900 4.5GHz processor and averaged over 100 different runs with random initial conditions.

Given that the execution horizon, T_h, is 2.5 seconds, our executive planner is able to generate local control trajectories in real time. Even in the worst case (three vehicles and ten regions), the optimization time took only 2.22 seconds. Note that the full mission time, T_M, for these scenarios ranges from a few minutes to several hours, demonstrating that our approach is applicable to a wide range of mission horizons.

In summary, our executive planner out performs the naive approach in situations where later activities impact early motion but still performs well on simple scenarios. Additionally, even though our MISOCP trajectory optimization encoding is more complicated than a naive approach, our executive planner runs in real-time on relatively large problems.

6 Conclusion & Future Work

In this paper, we presented an executive planner capable of executing complex multi-vehicle mission plans with multiple time-evolved goals. We use a discrete time formulation to plan local trajectories and a continuous time formulation to represent a relaxed solution for the full mission plan. Our key innovation is integrating these two models into a single mixed integer second-order cone program (MISOCP) which is solved iteratively using a receding horizon. We show that our approach ensures that local control trajectories are consistent with and guided by the constraints of the full plan and is broadly applicable to a range of complex robotic mission scenarios. Ongoing work aims to integrate our executive planner with a generative activity planner to create a hierarchical continuous planning and execution architecture.



(a) AUV + Ship (b) Hazardous Region (c) Motivating Scenario

Figure 5: Example executed plans for plan quality comparison scenarios (single AUV scenario not shown). The AUV is in orange and the ship is in blue.

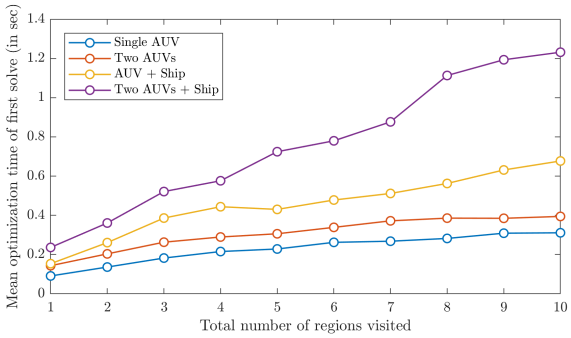


Figure 6: Performance of our executive planner evaluated on scenarios with up to three vehicles. In all scenarios, $\Delta t = 0.25$ seconds, $T_P = 8$ seconds, and $T_h = 2.5$ seconds.

References

Bellingham, J.; Kuwata, Y.; and How, J. 2003. *Stable Receding Horizon Trajectory Control for Complex Environments*. American Institute of Aeronautics and Astronautics.

Bhattacharya, S.; Likhachev, M.; and Kumar, V. 2010. Multi-agent path planning with multiple tasks and distance constraints. In *2010 IEEE International Conference on Robotics and Automation*, 953–959.

Cowlagi, R. V., and Tsiotras, P. 2012. Hierarchical motion planning with kinodynamic feasibility guarantees: Local trajectory planning via model predictive control. In *2012 IEEE International Conference on Robotics and Automation*, 4003–4008.

Ding, X. D.; Englot, B.; Pinto, A.; Speranzon, A.; and Surana, A. 2014. Hierarchical multi-objective planning: From mission specifications to contingency management. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, 3735–3742.

Fernández-González, E.; Williams, B. C.; and Karpas, E. 2018. ScottyActivity: Mixed discrete-continuous planning with convex optimization. *Journal of Artificial Intelligence Research* 62:579–664.

Hofmann, A. G., and Williams, B. C. 2017. Temporally and spatially flexible plan execution for dynamic hybrid systems. *Artificial Intelligence* 247:266–294.

Jain, S., and Tsiotras, P. 2009. Sequential multiresolution trajectory optimization schemes for problems with moving targets. *Journal of Guidance, Control, and Dynamics* 32(2):488–499.

Kuwata, Y., and How, J. P. 2004. Stable trajectory design for highly constrained environments using receding horizon control. In *Proceedings of the 2004 American Control Conference*, volume 1, 902–907 vol.1.

Léauté, T., and Williams, B. C. 2005. Coordinating agile systems through the model-based execution of temporal plans. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 1*, AAAI'05, 114–120. AAAI Press.

Mettler, B.; Dadkhah, N.; and Kong, Z. 2010. Agile autonomous guidance using spatial value functions. *Control Engineering Practice* 18(7):773 – 788. Special Issue on Aerial Robotics.

Richards, A.; Schouwenaars, T.; How, J. P.; and Feron, E. 2002. Spacecraft trajectory planning with avoidance constraints using mixed-integer linear programming. *Journal of Guidance, Control, and Dynamics* 25(4):755–764.

Schillinger, P.; Bürger, M.; and Dimarogonas, D. V. 2018. Simultaneous task allocation and planning for temporal logic goals in heterogeneous multi-robot systems. *The International Journal of Robotics Research* 37(7):818–838.

Schouwenaars, T.; Moor, B. D.; Feron, E.; and How, J. 2001. Mixed integer programming for multi-vehicle path planning. In *2001 European Control Conference (ECC)*, 2603–2608.

Ulusoy, A.; Smith, S. L.; Ding, X. C.; and Belta, C. 2012. Robust multi-robot optimal path planning with temporal logic constraints. In *2012 IEEE International Conference on Robotics and Automation*, 4693–4698.

Wurm, K. M.; Dornhege, C.; Nebel, B.; Burgard, W.; and Stachniss, C. 2013. Coordinating heterogeneous teams of robots using temporal symbolic planning. *Auton. Robots* 34(4):277–294.

Monitoring Numeric Expectations in Goal Reasoning Agents

Noah Reifsnyder and Héctor Muñoz-Avila
Lehigh University
Bethlehem, PA 18015
{ndr217, hem4}@lehigh.edu

Abstract

One of the crucial capabilities for robust agency is self-assessment, namely the capability of the agent to compute its own boundaries. Goal reasoning agents do this by computing so called expectations: constructs defining the boundaries of their courses of action as a function of the plan, the goals achieved by that plan when available, the initial state, the action model and the last action executed. In this paper we introduce four forms of expectations when the agent reasons with numeric fluents and present an empirical evaluation highlighting their trade offs.

Introduction

Over the past years there has been an increasing interest in goal reasoning agents; agents that may change their goals over time as a result of changes in the environment and/or changes in the user's requirements (Aha 2018). One of the main motivations for goal reasoning is the robust intelligence problem, where agents exhibit the following capabilities (Tianfield and Unland 2004):

- Are aware of their own limits or boundaries (self-assessment) (Sloman and Logan 1999).
- Recognize when they have stepped out of those boundaries (self-diagnosis) (Horling, Benyo, and Lesser 2001).
- Act to bring themselves back into their boundaries (self-correction) (Lucas 1961).

In this paper we focus on the **self-assessment** problem of goal reasoning agents. Our worked is motivated by the premise that it is infeasible for the agent to plan ahead for every possible contingency that it may encounter when executing a course of action. On the other hand, it is also unfeasible to replan to every possible contingency. In the words of Ghallab et al (2014) *"Current approaches in the planning literature either tend to foresee all possible events and situations, which is unpractical in realistic complex domains, or they tend to replan any time something unexpected occurs, which is hard to do in practice at run-time"*.

In domains with numeric fluents, it has been observed that it is contrived to assume after performing each action, each fluent will be expected to have an exact value (Scala et al. 2016). For instance when traveling between two location that a precise amount of gasoline will be consumed by the vehicle. Even under "usual" conditions, factors such as traffic accidents may cause delays. Authors have observed that a more robust way to handle numeric fluents is to use intervals or margins of error (Moore, Kearfott, and Cloud 2009). This reduces replanning since the agent can plan accounting for variations avoiding the need for repeated replanning. Nevertheless, the agent may encounter conditions under which fluents may take values outside predefined ranges.

Numeric fluents can be altered by actions through functions instead of by simple assignment of values (Hoffmann 2003; Coles et al. 2010). This presents a challenge to the concept of expectations for goal reasoning agents. Applying actions change symbolic fluents in the usual way (e.g., using add and delete lists) and change numeric fluents, representing a numeric value v, to a new value $f(v)$ as indicated by the actions' effects (Gerevini, Saetti, and Serina 2008). For example, for a *navigate* action, if $v = (energy\ ?x)$, then $f(v) = v - (t * r)$, where t = *(travel-time ?y ?z)* and r =*(use-rate ?x)*. The starting state, s_0 includes numeric fluents such as the fuel level of *rover21*, travel time between locations, and symbolic fluents such as the starting location of *rover21*. Frequently, for planning purposes these functions are assumed to be monotonic (Edelkamp 2003; Hoffmann 2003; Bajada, Fox, and Long 2015) although some paradigms drop this assumption (Scala et al. 2016).

In this paper we reexamine a taxonomy that encompasses expectations as computed by goal reasoning systems for symbolic fluents and extend that taxonomy for numeric fluents and their margins of error (Munoz-Avila, Dannenhauer, and Reifsnyder 2019). The taxonomy's starting point is the division of the plan $\pi = a_1..a_n$ generated between two parts $\pi_{prefix} = a_1..a_i$, the actions in π already executed and $\pi_{suffix} = a_{i+1}..a_n$, the actions in π to execute. We redefine the four forms of expectations from the goal reasoning literature for the numeric case: immediate, which checks the effects of the last action, a_i executed; goal-regression, which computes the conditions needed to execute π_{suffix}; informed, which accumulated the effects from actions in π_{prefix}; and Goldilocks, which combines informed and regression expectations.

Up until now, the bulk of the research on goal reasoning has focused on domains where actions have symbolic fluents. The only exceptions we know are (Weber, Mateas, and Jhala 2012) and (Wilson, McMahon, and Aha 2014); these use immediate and informed expectations respectively.

The following are the main contributions of this paper:

- We extend the taxonomy of expectations in (Munoz-Avila, Dannenhauer, and Reifsnyder 2019) for plans with numeric fluents and margins of error.
- We analyze the tradeoffs between those expectations.

(:operator move_north :parameters ?r :condition not-within(at-y(?r), [0, 1]), within(fuel(?r), [right(rate(?r)), ∞)) :effect at-y(?r) = [f_1(?r), f_2(?r)], fuel(?r) = [f_3(?r), f_4(?r)], f_1(x) = left(at-y(x))-1, f_2(x) = right(at-y(x))-1, f_3(x) = left(fuel(x)) - right(rate(x)), f_4(x) = right(fuel(x)) - left(rate(x))

Table 1: Example of operator with a numeric fluent (fuel)

(:Initial State {fuel : {r23: [10,10]}} {at-y : {r23: [2, 2], Beacon1: [0,0]}} {at-x : {r23: [0,0], Beacon1: [2,2]}} {lit : {Beacon1: [0, 0]}} {rate : {r23: [.9, 1.1]}} :Actions move_north, move_south, move_east, move_west, light_beacon :Goals {lit : {Beacon1: [1, 1]}} :Plan π move_north, move_north, move_east, move_east, light_beacon

Table 2: Planning problem and a solution plan

- We perform experiments on a goal reasoning system with the four forms of expectations.

Our work doesn't make any assumptions of how the plan π was generated.

States, Operators and Plans with Numeric Fluents

A state is a collection of variables that can be either symbolic or numeric. In this paper we will focus on actions with numeric fluents for simplicity of the exposition. A state is a mapping $s : V \rightarrow \mathbb{I}$ from a collection of variables V to a collection of intervals $\mathbb{I}$. For instance, the variable *at-y(r23)* returns the y-coordinate as a confidence interval *[left(at-y(r23)),right(at-y(r23))]* for rover *r23*.

As exemplified in Table 1, an operator is a 4-tuple *o=(name parameters precondition effect)*. The parameters are a collection of free variables. We call free variables to variables that are not variables in the state. Free variables are used to facilitate writing operators. We denote free variables by using "?". For instance *?x* denotes the free variable x. The precondition is a set of numeric fluents and interval constraints where the preconditions can be represented as a partial mapping $pre : V \nrightarrow \mathbb{I}$. A fluent nf is either represented as a closed interval $[left(nf), right(nf)]$, with $left(nf)$ and $right(nf)$ numbers and $left(nf) \leq right(nf)$; or if either side is unbounded, that bound is represented as an open interval bound on ∞ (or $-\infty$). Interval constraints are of the form *(within nf interval)* where nf is a fluent. Within checks if $left(nf) \geq left(interval)$ and $right(nf) \leq right(interval)$. For instance, in the operator shown in Table 1 we are checking if the interval for the numeric fluent *fuel(?r)* is within the interval left bounded by $right(rate(?r))$ and right bounded by ∞. The effects indicate changes in value to the state variable. These are represented as function tuples $\mathbb{F} = \{(f_1, f_2) | f_1 \text{ and } f_2 \text{ are functions}\}$. The effects can be represented as the partial mapping $eff : V \nrightarrow \mathbb{F}$, where each variable $v \in V_{eff}$ therefore has the function tuple (f_1^v, f_2^v).

An action takes a variable $v \in V$ and assigns v value $[f_1(left(v)), f_2(right(v))]$, where $(f_1, f_2) \in \mathbb{F}$ and $f_1(x) <= f_2(x)$ for all x. For example, *fuel(?r)* represents the fuel level of vehicle r. Table 1 shows an example of an operator that uses *fuel(?r)*. The action $move\_north$ takes the variable $fuel(?r)$ and alters it with the following functions $fuel(?r) = [left(fuel(?r)) - right(rate(?r)), right(fuel(?r)) - left(rate(?r))]$. This denotes the usage of fuel by the action $move\_north$. The interval of rate(?r) represents amount of fuel we expect to be consumed by moving a tile. By subtracting the maximum of rate from the minimum of our current fuel level, we get the new minimum amount of fuel we expect to have after executing $move\_north$. By subtracting the minimum of the rate from the maximum of our current fuel level, we get the new maximum amount of fuel we expect to have. $move\_north$ also alters the variable at-y(?r). at-y(?r) is altered by the function tuple at-y$(?r) = (left($at-y$(?r)) - 1, right($at-y$(?r)) - 1)$. This denotes a change in y-position of the rover of -1 (north). So all together, the action $move\_north$ consumes the interval of rate(?r) amount of fuel while moving -1 in the y-coordinate plane.

Two Basic Operations

We introduce two basic operations $\oplus_S$ and $\ominus_P$, which are used to define precisely the different forms of expectations.

We define $D = A \oplus_S B$, where A are some variables, S is the current state and B are the effects of some action. More generally, for any partial functions A and B and and any function S with $A : V \nrightarrow \mathbb{I}$, $S : V \rightarrow \mathbb{I}$, and $B : V \nrightarrow \mathbb{F}$, $A \oplus_S B$ is a partial mapping $D : V \nrightarrow \mathbb{I}$ defined as follows:

1. if $v \in V_A \cap V_B$ and $B(v) = (f_1, f_2)$, then $D(v) = [f_1(left(A(v))), f_2(right(A(v)))]$.
2. if $v \in V_A - V_B$ then $D(v) = A(v)$.
3. if $v \in V_B - V_A$ where $B(v) = (f_1, f_2)$, then $D(v) = [f_1(left(S(v))), f_2(right(S(v)))]$.
4. for all other variables D is undefined (i.e., $V_D = V_A \cup V_B$)

Informally, $A \oplus_S B$ applies the function tuple in B either to A when the variable v is defined in A and B (Case 1), or to S when v is defined in B but not A (Case 3). If the variable v is defined in A but not B, it's assigned $A(v)$ (Case 2). When it's undefined in A and B, then it's left undefined (Case 4). For example, if A, S, and B are defined as:

- $A = \{a : [2, 3]\}$
- $S = \{a : [2, 3], b : [7, 7], c : [8, 9], d : [6, 6]\}$

- $B = \{a : [x-2, x-1], b : [x+1, x+2], d : [x \times 2, x \times 3]\}$
- Then $D = A \oplus_S B = \{a : [0, 2], b : [8, 9], d : [12, 18]\}$.

In the resulting partial function $D(a) = [0, 2]$ is obtained by evaluating the functions tuple $B(a) = [x-2, x-1]$ on the interval $A(a) = [2, 3]$ (Case 1); $D(b) = [9, 11]$ is obtained by evaluating the functions tuple $B(b) = [x + 1, x + 2]$ on the value of $S(b) = [7, 7]$ (Case 3); and $D(d) = [12, 18]$ is obtained by evaluating the functions tuple $B(d) = [x \times 2, x \times 3]$ on the value of $S(d) = [6, 6]$ (Case 3).

We define $D = A \ominus_P B$, where A are some variables, P are the preconditions from some action and B are the effects of the action. More generally, let $A : V \nrightarrow \mathbb{I}$, $P : V \nrightarrow \mathbb{I}$, and $B : V \nrightarrow \mathbb{F}$, we define $A \ominus_P B$ as a partial mapping $D : V \nrightarrow \mathbb{I}$ with:

1. if $v \in V_A - V_B$ then $D(v) = A(v)$.
2. if $v \in (V_A \cap V_B)$ and $B(v) = (f_1, f_2)$, then $D(v) = [f_1^{-1}(left(A(v))), f_2^{-1}(right(A(v)))]$.
3. if $v \in V_P - V_A$ then $D(v) = P(v)$
4. for all other variables D is undefined (i.e., $V_D = V_A \cup V_P$)

Informally, $A \ominus_P B$ results in a new partial mapping that is defined for all variables from A and P. The new mapping takes the value $A(v)$ if v is defined in A but not in B (Case 1). If a variable v is defined in A and B, the new mapping takes the values after applying the inverse of the functions tuple defined in $B(v)$ to the value of $A(v)$ (Case 2). If a variable v is defined in P but not in A, the new mapping takes the value of $P(v)$ (Case 3). If a variable is not defined in either A or P, it is left undefined (Case 4). For example, if we have the three partial functions A, P, and B, as follows:

- $A = \{a : [2, 3], b : [5, 6]\}$
- $P = \{c : [4, 4]\}$
- $B = \{b : [x + 1, x + 2]\}$
- Then $D = A \ominus_P B = \{a : [2, 3], b : [4, 4], c : [4, 4]\}$.

In the resulting function, $D(a) = [2, 3]$ because a is defined in A but not in B (Case 1); $D(b) = [4, 4]$ because $A(b) = [5, 6]$ and $B(b) = [x + 1, x + 2]$, hence the inverse functions are $[x - 1, x - 2]$ (Case 2); and $D(c) = [4, 4]$ because c is defined in P but not in A (Case 3).

Immediate Expectations with Numeric Values

Immediate Expectations takes ideas from plan monitoring execution literature (see related work discussion). Formally, $X_{imm}(\pi, s_i, \emptyset) = imm_i$. Each imm_i is generated as follows: $imm_0 = pre^{a_1}$. For all $i > 0, imm_i = (pre^{a_{i+1}} \ominus_{\{\}} eff_{s_i}^{a_i}) \oplus_{s_{i-1}} eff_{s_i}^{a_i}$. Informally, agents using immediate expectations check that the effects of the previous action a_i and the preconditions of the next action to execute a_{i+1} hold in the observed state s_i (Cox 2007).

Example:

In the plan π in Table 2, assume we have completed a_1 = move-north and we are about to execute a_2 = move-north, then the Immediate Expectations, $imm_1 = (pre^{move\_north} \ominus_{\{\}} eff_{s_1}^{move\_north}) \oplus_{s_0} eff_{s_1}^{move\_north}$ with:

- $pre^{move\_north} = (\{fuel : \{r23 : [1.1, \infty]\}\}$
- $eff_{s_1}^{move\_north} = \{at\text{-}y : \{r23 : [x - 1, x - 1]\}, fuel : \{r23 : [x - 1.1, x - .9]\}\}$
- Then $(pre^{move\_north} \ominus_{\{\}} eff_{s_1}^{move\_north}) = \{fuel : \{r23 : [2.2, \infty]\}\}$
- and $imm_1 = (pre^{move\_north} \ominus_{\{\}} eff_{s_1}^{move\_north}) \oplus_{s_0} eff_{s_1}^{move\_north} = \{at\text{-}y : \{r23 : [1, 1]\}, fuel : \{r23 : [1.1, \infty]\}\}$

Since $fuel(r23)$ is a variable in common (Case 2 of the $\ominus_{\{\}}$ operation), we apply the inverse of $fuel : \{r23 : [x - 1.1, x-.9]\}$ which is $fuel : \{r23 : [x+1.1, x+.9]\}$ to $fuel : \{r23 : [1.1, \infty]\}$. The resulting value for $fuel(r23) = [(x+1.1)(1.1), (x + .9)(\infty)) = [2.2, \infty)$

Finally, $X_{imm}(\pi, s_1, \emptyset) = \{fuel : \{r23 : [0, \infty]\}\} \oplus_{s_0} eff_{s_1}^{move\_north} = \{at\text{-}y : \{r23 : [1, 1]\}, fuel : \{r23 : [1.1, \infty]\}\}$ because $fuel(r23)$ is a common variable on the left and right side of $\oplus_{s_0}$ (Case 1 of the $\oplus_{s_0}$ operation) and *at-y(r23)* is a common variable in s_0 and $eff_{s_1}^{move\_north}$ (Case 3 of the $\oplus_{s_0}$ operation). Thus $fuel(r23) = [((x - 1.1)(2.2), (x - .9)(\infty)) = [1.1, \infty)$, and $(at\text{-}y(r23) = [(x-1)(2), (x-1)(2)] = [1, 1]$. This expectation set means that we expect to have at least 1.1 units of fuel and to be at y=1 on the coordinate frame (any x coordinate is fine)

Informed Expectations with Numeric Values

Informed Expectations continuously build on effects from all previous actions executed so far in π. Informally, it compounds functions tuples $[left(v"), right(v")] = [f_1(\dots(f_1'(left(v)))\dots), f_2(\dots(f_2'(right(v)))\dots)]$, where (f_1', f_2') are the effects for v of the first action in the trace and (f_1, f_2) are the effects for v of the last action executed. If v is not changed in some action a" then we assume $f_1"(x) = f_2"(x) = x$ (i.e., the identity function). Formally, $X_{inf}(\pi, s_i, \emptyset) = inf_i$. Each inf_i is generated as follows: $inf_0 = \emptyset$. That is, before the first action is executed, we have no accumulated effects. For $i > 0$, inf_i is defined recursively as follows: $inf_i = inf_{i-1} \oplus_{s_{i-1}} eff_{s_i}^{a_i}$. Agents using Informed Expectations check that the compounded effects are valid in the environment.

Example:

If we have just completed action a_2 of the plan trace π (the second instance of $move\_north$), we calculate the Informed Expectations $X_{inf}(\pi, s_2, \emptyset) = inf_2$ as follows from the initial state. We have:

- $inf_1 = \{at\text{-}y : \{r23 : [1, 1]\}, fuel : \{r23 : [8.9, 9.1]\}\}$
- inf_2 compounds inf_1 with the effects from a_2, the second $move\_north$:
  - $eff_{s_2}^{move\_north} = \{at\text{-}y : \{r23 : [x-1, x-1]\}, fuel : \{r23 : [x - 1.1, x - .9]\}\}$
- Thus, $inf_2 = inf_1 \oplus_{s_1} eff_{s_2}^{move\_north} = \{at\text{-}y : \{r23 : [0, 0]\}, fuel : \{r23 : [7.8, 8.2]\}\}$

For computing *at-y(r23)* we compute $[(x - 1)(1), (x - 1)(1)] = [0, 0]$ and for *fuel(r23)* we compute $[(x - 1.1)(8.9), (x - .9)(9.1)] = [7.8, 8.2]$. This expectation set

means that we expect to have between 7.8 and 8.2 units of fuel and to be at y=0 on the coordinate frame (any x coordinate is fine)

Regression Expectations with Numeric Values

Regression Expectations continuously build on the cumulative values of the regressed conditions from all actions yet to be executed in π. Informally, it compounds functions tuples $[left(v'), right(v')] = [f_1^{-1}(\dots(f_{1'}^{-1}(left(v)))\dots), f_2^{-1}(\dots(f_{2'}^{-1}(right(v)))\dots)]$, where $(f_{1'}^{-1}, f_{2'}^{-1})$ are the inverse of the effects for v of the last action in the trace and (f_1^{-1}, f_2^{-1}) are the inverse of the effects for v of the next action to be executed. Formally, $X_{regress}(\pi, s_i, \mathcal{G}) = reg_i$. Each reg_i is generated as follows: $reg_n = \mathcal{G}$. That is, when in the last state, the agent expects $\mathcal{G}$ to hold (when the goals are unknown, $\mathcal{G}$equals the empty set $\{\}$). For $i < n$, reg_i is defined recursively as follows: $reg_i = reg_{i+1} \ominus_{pre^{a_{i+1}}} eff_{s_{i+1}}^{a_{i+1}}$. Informally, the agent checks the needed preconditions to execute the remaining of the plan $a_{i+1} \dots a_n$ while having the goals $\mathcal{G}$ satisfied in s_n.

Example:

If we have just completed action a_3 of the plan trace π in Table 2 (the first instance of $move\_east$), we calculate the Regression Expectations $reg_3 = X_{regress}(\pi, s_3, \mathcal{G})$ as follows (The preconditions and effects for $move\_east$ and $light\_beacon$ have not been shown before):

- $reg_5 = \mathcal{G} = \{lit : \{Beacon1 : [1,1]\}\}$ i.e. the Goals.
- $reg_4 = \{at{-}y : \{r23 : [0,0]\}, at{-}x : \{r23 : [2,2\}, lit : \{Beacon1 : [0,0\}\}$ which are the preconditions for $a_5, light\_beacon$.
- $reg_3 = reg_4 \ominus_{pre^{a_4}} eff_{s_4}^{a_4}$, where:
  - $pre^{a_4} = \{at{-}x : \{r23 : [0,1]\}, fuel : \{r23 : [1.1, \infty)\}\}$
  - $eff_{s_4}^{a_4} = \{at{-}x : \{r23 : [x+1, x+1]\}, fuel : \{r23 : [x-1.1, x-.9]\}\}$
- Thus, $reg_3 = \{at{-}y : \{r23 : [0,0]\}, at{-}x : \{r23 : [1,1]\}, fuel : \{r23 : [1.1, \infty)\}, lit : \{Beacon1 : [0,0\}\}$

reg_4 follows from the effects of $light\_beacon$, which increase $lit(Beacon1)$ by $(x+1, x+1)$ so the inverse generates $lit(Beacon1) = [(x-1)(1), (x-1)(1)] = [0,0]$. Therefore, $reg_3 = reg_4 \ominus_{pre^{move\_east}} eff_{s_4}^{move\_east}$, where a_4 is the second instance of $move\_east$. Since $fuel(r23)$ is in pre^{move_east} and not in reg_4, in reg_3: $fuel(r23) = pre^{move\_east}(fuel(r23)) = [1.1, \infty)$. Since $at{-}y(r23)$ and $lit(Beacon1)$ are in reg_4 and not in $eff_{s_4}^{move\_east}$, they just carry over from reg_4 into reg_3. $at{-}x(r23)$ is in both, reg_4 and $eff_{s_4}^{move\_east}$, so we compute the inverse functions from $eff_{s_4}^{move\_east}$ to get in reg_3: $at{-}x(r23) = [(x-1)(2), (x-1)(2)] = [1,1]$. This expectation set means that we expect to have at least 1.1 units of fuel, to be at y=0 and x=1 on the coordinate frame, and for the beacon to not be lit.

Goldilocks Expectations with Numeric Values

Goldilocks Expectations (Reifsnyder and Munoz-Avila 2018) combines Informed and Regression Expectations. They are computed by calculating informed expectations from the starting state (i.e., in the numeric case by compounding functions $v = f(\dots(f'(v'))\dots)$), then regressing off from the final states the informed expectations (i.e., in the numeric case by compounding inverse functions $v'' = f'^{-1}(\dots(f^{-1}(v))\dots)$). This would not work in the numeric case since it will regress to the exact same value (i.e., $v" = v'$). We present a slightly different definition for Goldilocks Expectations; it takes into account both Informed and Regression Expectations independently.

The reason for doing this is because Regression can detect when the agent will not achieve its goals, while Informed can detect that something is wrong in the execution of the plan. By looking at them independently, the agent can make decisions over trade-offs between achieving the goals and how the agent is achieving them. For example, using the navigation domain from Table 2, an agent might have a goal to end the plan trace π with a range of fuel left, e.g., $\{fuel : \{r23 : [0, \infty)\}\}$. In this scenario, Regression Expectations will make sure the agent has enough fuel to finish π. While this is important, it is also important to realize that Informed Expectations keep track of accumulated effects of the the actions executed so far. Informed is monitoring here the fuel consumption of the agent, and making sure it remains within the bounds as inferred from the action model. If the agent drifts out of those bounds, there may be a flaw with the agent causing it to consume more fuel than projected. So the agent might still achieve the goals but consume more fuel than expected. Recognizing this expectation failure can allow the agent to trigger a discrepancy and avoid needlessly wasting fuel. By considering both of these expectations, the agent can detect a variety of possible failures at their onset beyond "just" achieving the goals.

Formally, we define Goldilocks Expectations as $X_{gold}(\pi, s_i, \mathcal{G}) = gold_i$, where $gold_i = (inf_i, reg_i)$. That is, for ever state s_i, $gold_i$ is the pair containing the Informed and Regression Expectations for that state. An agent using $X_{gold}(\pi, s_i, \mathcal{G})$ checks the overlap of the regressed and the informed intervals, $[left(v'), right(v')] \cap [left(v"), right(v")]$. This ensures completing the goals while checking for inferred considerations from the action model such as efficiency.

Example:

When the agent completes action a_3 of the plan trace π in Table 2 (the first instance of $move\_east$), it calculates the Goldilocks Expectations $gold_3$ as follows. $gold_3 = (inf_3, reg_3)$, both of which we exemplified previously as:

- $inf_3 = (\{at{-}y : \{r23 : [0,0]\}, at{-}x : \{r23 : [1,1]\}, fuel : \{r23 : [6.7, 7.3]\}\}$
- $reg_3 = \{at{-}y : \{r23 : [0,0]\}, at{-}x : \{r23 : [1,1]\}, fuel : \{r23 : [1.1, \infty)\}, lit : \{Beacon1 : [0,0]\}\})$

There is a difference in the two expectations over the expectations computed for variable $fuel(r23)$. On the In-

formed side we expect between 6.7 and 7.3 fuel units, but on the Regression side, we just expect to have more than 1.1 fuel units. If we violate the Informed side but not the Regression side, we know we can likely finish the plan, but it will indicate a larger than expected fuel consumption.

Empirical Evaluation

In our experiments, we tested 4 different types of expectations across numeric extensions of 2 domains used in the goal reasoning literature. The 4 Expectation types we tested were Immediate, Informed, Regression, and Goldilocks Expectations. For planning purposes, we use the Pyhop HTN planner (Nau 2013), which handles numeric fluents. Other than the expectation type, the agent uses the same planning and discrepancy handling processes. Whenever a discrepancy is observed from the expectations, we use a simple goal-reasoning process to generate a goal to re-plan from the current state.

Marsworld Definition The first domain we used is a variant on the domain Marsworld (Dannenhauer, Munoz-Avila, and Cox 2016; Dannenhauer and Munoz-Avila 2015), inspired by Mudsworld (Molineaux and Aha 2014). The agent has to navigate a 10x10 grid to turn on 3 randomly placed beacons. Each movement action drains some amount of the agents fuel, which is determined by a predetermined rate for the agent. The agent also has known error rate for consuming fuel. Lighting each beacon also requires fuel, and consumes fuel from a different reserve then from where the agent draws from for movement.

While executing its actions, the agent may unexpectedly have damage caused to it, forcing it to use more fuel per action until repaired (this can occur with a 5% probability after each action is taken). It can also lose some of its beacon fuel with a 5% probability after each action as well. During our testing, we ran 100 trials, each trial placed the rover and beacons randomly on the grid. During the trials we measured total fuel consumption as well as whether or not an execution failed. A failure means the preconditions of some action were not met when it was to be executed.

Results for Marsworld. In Figure 1, we can see that Regression Expectations consumed the most fuel, with the 3 other expectation types performing basically equally. The reason for this, is the Regression Expectations are the only ones not noticing when the agent is damaged, causing increased fuel consumption. Regression only looks at future preconditions, so it only realizes the damage once it drains enough fuel so that it no longer has enough to finish its plan. The other 3 expectation types identify increase consumption after 1 action, since they monitor effects of the actions.

Figure 2 shows the error rates for each expectation type Immediate, Regression and Goldilocks are able to ensure that the plan will be completed without failures, while 27% of trials failed for Informed Expectations. Informed fails because it will attempt to execute an action without it's preconditions being met. All other expectation types check preconditions. Specifically, in this scenario, agents using Informed expectations will attempt to light a beacon after having lost some beacon fuel, thus failing the action.

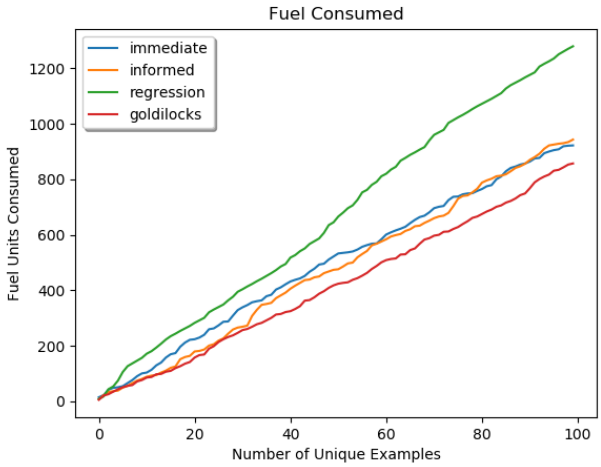


Figure 1: Accumulated Fuel Consumption in Marsworld across different types of expectations

Blockscraft Definition. The second domain we tested in is a variant of the Blockscraft domain (Dannenhauer, Munoz-Avila, and Cox 2016). In our variant, there are three towers of blocks; each block has a random mass. The mass of each block can only be estimated at planning time, so while the exact mass is unknown, a range for the mass is known and the exact mass is guaranteed to be within that range. We have an estimation of every blocks' mass. The agent can only access the top block in each tower, and can only know the exact mass of the block after collecting it. The task for the agent is to create a tower of blocks and the tower as a whole must be greater than a certain mass.

There is another actor in the environment who can take blocks both from the 3 towers our agent is using, as well as from the agent's own tower. There is an 8% chance after each action that the other actor will take a block out of the 3 center towers, and a 2% chance that the actor takes from the agent's tower. We ran 100 trials, where the mass of the blocks were randomized each trial. We measured total mass obtained by the agent during each trial, as well as if the trial failed or not. A failure happens either if the agent finishes the plan without enough mass in their tower, or an action's preconditions are not met when it is to be executed.

Results for Blockscraft. Figure 3 shows the total mass accumulated across 100 runs for an agent using each type of Expectations. Immediate, Informed, Regression, and Goldilocks all accumulated roughly the same amount of mass. They are all equal, because there isn't any discrepancies that alters the rate of obtaining mass, so the rates stay constant between all expectation types.

Figure 4 shows the failure rates. Only an agent using Goldilocks Expectations was able to complete the plan and have the goals fulfilled. Immediate failed 65% of its trials, while Regression failed 43% of its trials. These failures occurred when the other actor took blocks out of the agent's tower. Effects of actions are not monitored for those expectations, so the agent is not monitoring total mass of its tower.

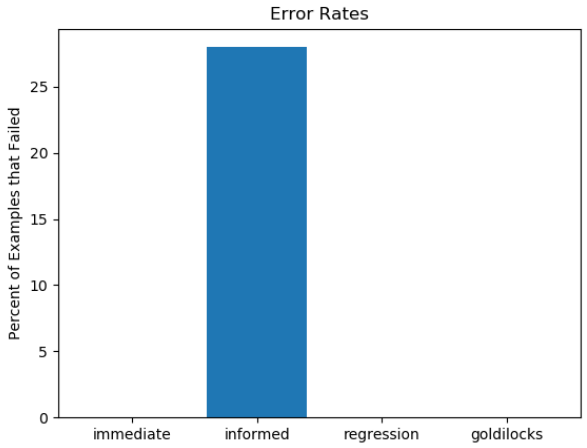


Figure 2: Number of failures in 100 executions in Marsworld across different types of expectations

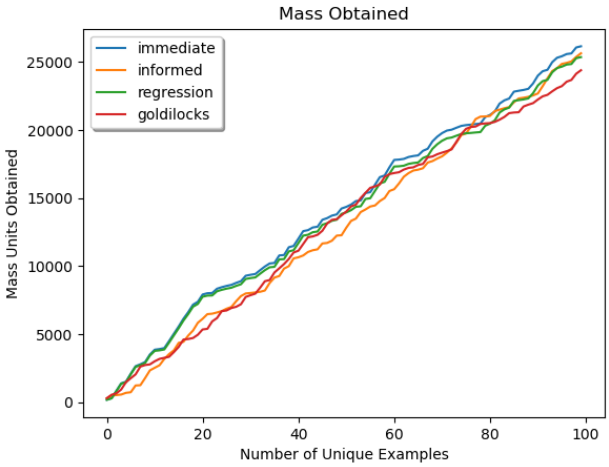


Figure 3: Accumulated Mass Obtained in the Blocks World Domain across different types of expectations

Informed failed 88% of its trials. This occurred due to the other actor removing blocks from the 3 central towers that the agent had planned to obtained. The action then to collect that block failed because the block didn't exist in the tower any more. Goldilocks had 0 failures across the 100 trials. This is because it checked preconditions due to the Regression side of the expectations, while also monitoring the mass so far obtained by the agent from the Informed side of the expectations. Combined, an agent using Goldilocks Expectations caught all discrepancies.

Related Work

(Scala 2013) proposed using kernel methods to compute the necessary numerical conditions K^i needed to complete the rest of the plan $a_i \dots a_n$, akin to our regression expectations. While not defined that way, $K^i(v) = f_i^{-1}(\dots(f_n^{-1}(v))\dots)$, where v is a goal condition and f_j^{-1} is the inverse function for v in a_j (with $i \leq j \leq n$). (Scala and Torasso 2014) expands this to distances around the values of v playing a similar role as our error intervals. Our work differs in some important differences: by explicitly using inverse functions we provide a concrete way to compute these kernels. More importantly, we introduce two forms of expectations: informed and Goldilocks. Informed expectations are needed when goals are not known. Informed expectations can provide needed information missing from goal regression calculations. For example, in a scenario where we have resource consumption, if we have an action that consumes an amount of resource, we would likely have some goal to have > 0 amount of the resource (or more than however much the action consumes). If there are multiple occurrences of this action, and they end up consuming a larger amount of resource, informed will allow us to (1) detect the discrepancy after the first action, instead of after however many it takes to deplete the resource and (2) conserve more of the resource. Crucially, Goldilocks allows the detection of

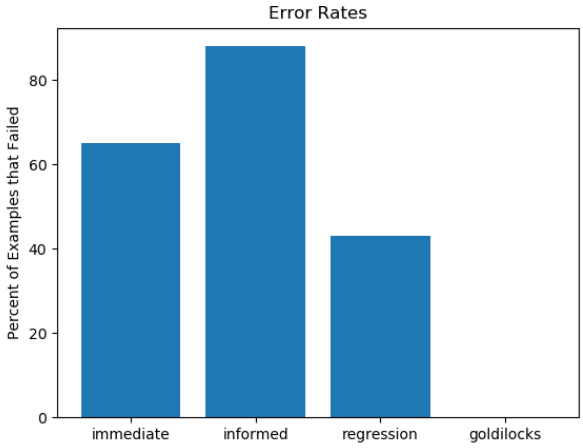


Figure 4: Number of failures in 100 executions in Blocks World Domain across different types of expectations

deviations in the values of numerical values that are pivotal in finishing the provided plan with the expected outcomes, even when the goals are unknown.

We know of two systems using numeric fluents for goals reasoning. (Weber, Mateas, and Jhala 2012) represents quantities as numeric fluents for a goal reasoning agent playing an adversarial real-time computer game. For instance, an action to *produce 10 archers*, will have as expectation that 10 archers are produced. After executing the action, if the number of archers is num with $num < 10$, then a discrepancy is detected. The agent will formulate a new goal to produce $10 - num$ archers. In the context of the taxonomy we presented, this agent maintains immediate expectations. Furthermore, no margins of error are maintained as the agents expectations are exact natural numbers.

(Wilson, McMahon, and Aha 2014) uses what we call informed expectations. They project forward the expected

numerical values within intervals and detect discrepancies when the values are outside of these projected intervals. Our experiments show how informed expectations can incur into the highest number of errors because they don't regress conditions on the goals. This leads to a high volume of failures when they attempt to be executed.

Plan monitoring execution systems annotate the plan with conditions necessary for the plan's execution to be valid (Fikes, Hart, and Nilsson 1972). While not using "expectations" as a term, (Ambros-Ingerson and Steel 1988) checks for the causal links, triples *(effect, fluent, precondition)* are met when the action having the precondition is to be executed. In our parlance, this is subsumed by immediate expectations. Plan monitoring execution have also been used to monitor optimality. For instance, (Fritz and McIlraith 2007) uses goal regression to define necessary conditions to guarantee the optimal execution of the plan. These works use symbolic fluents.

Conclusions

We introduce 4 forms of Expectations over numerical fluents: Immediate, Informed, Regression, and Goldilocks. In our empirical evaluation, only agents using Goldilocks solved problems without failures across all domains; Goldilocks projects forward all changes the agent makes to the state, as well as making sure the agent is on track to meet the goals. The agent using Regression was shown to be inefficient over the fuel consumption in one of the domains. The reason is that Regression only checks if the agent is on track to satisfy the goals but makes no consideration of any other deviation from the action model.

Time series have been used to build statistical models of "normal" or expected readings for numerical values and in doing so detect outliers to predict malfunctions (Tsay 1988). In this work, we assume the "normal" ranges has been given in the effects of the actions. However, in future work, such models could be used in situations when we expect the effects of the actions to change over time.

Acknowledgements. This research was supported by ONR under grants N00014-18-1-2009 and N68335-18-C-4027.

References

Aha, D. W. 2018. Goal reasoning: foundations emerging applications and prospects. *AI Magazine*.

Ambros-Ingerson, J. A., and Steel, S. 1988. Integrating planning, execution and monitoring. In *AAAI*, volume 88, 21–26.

Bajada, J.; Fox, M.; and Long, D. 2015. Temporal planning with semantic attachment of non-linear monotonic continuous behaviours. In *IJCAI*, 1523–1529.

Coles, A. J.; Coles, A. I.; Fox, M.; and Long, D. 2010. Forward-chaining partial-order planning. In *Twentieth International Conference on Automated Planning and Scheduling*.

Cox, M. T. 2007. Perpetual self-aware cognitive agents. *AI magazine* 28(1):32.

Dannenhauer, D., and Munoz-Avila, H. 2015. Raising expectations in gda agents acting in dynamic environments. In *IJCAI*, 2241–2247.

Dannenhauer, D.; Munoz-Avila, H.; and Cox, M. T. 2016. Informed expectations to guide gda agents in partially observable environments. In *IJCAI*, 2493–2499.

Edelkamp, S. 2003. Taming numbers and durations in the model checking integrated planning system. *Journal of Artificial Intelligence Research* 20:195–238.

Fikes, R. E.; Hart, P. E.; and Nilsson, N. J. 1972. Some new directions in robot problem solving. *Machine Intelligence* 7:405–430.

Fritz, C., and McIlraith, S. A. 2007. Monitoring plan optimality during execution. In *ICAPS*, 144–151.

Gerevini, A. E.; Saetti, A.; and Serina, I. 2008. An approach to efficient planning with numerical fluents and multi-criteria plan quality. *Artificial Intelligence* 172(8-9):899–944.

Hoffmann, J. 2003. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research (JAIR)* 20:291–341.

Horling, B.; Benyo, B.; and Lesser, V. 2001. Using self-diagnosis to adapt organizational structures. In *Proceedings of the fifth international conference on Autonomous agents*, 529–536. ACM.

Lucas, J. R. 1961. Minds, machines and gödel. *Philosophy* 36(137):112–127.

Molineaux, M., and Aha, D. W. 2014. Learning unknown event models. In *AAAI*, 395–401.

Moore, R. E.; Kearfott, R. B.; and Cloud, M. J. 2009. *Introduction to interval analysis*, volume 110. Siam.

Munoz-Avila, H.; Dannenhauer, D.; and Reifsnyder, N. 2019. Is everything going according to plan? - expectations in goal reasoning agents. In *Proceedings of AAAI-19*.

Nau, D. 2013. Pyhop, version 1.2.2 a simple htn planning system written in python. `https://bitbucket.org/dananau/pyhop`. Accessed: 2019-01-30.

Reifsnyder, N., and Munoz-Avila, H. 2018. Goal reasoning with goldilocks and regression expectations in nondeterministic domains. In *6th Goal Reasoning Workshop at IJCAI/FAIM-2018*.

Scala, E., and Torasso, P. 2014. Proactive and reactive reconfiguration for the robust execution of multi modality plans.

Scala, E.; Haslum, P.; Thiébaux, S.; and Ramirez, M. 2016. Interval-based relaxation for general numeric planning. In *Proceedings of the Twenty-second European Conference on Artificial Intelligence*, 655–663. IOS Press.

Scala, E. 2013. Numeric kernel for reasoning about plans involving numeric fluents. In *Congress of the Italian Association for Artificial Intelligence*, 263–275. Springer.

Sloman, A., and Logan, B. 1999. Building cognitively rich agents. *Communications of the ACM* 42(3):71–72.

Tianfield, H., and Unland, R. 2004. Towards autonomic

computing systems. *Engineering Applications of Artificial Intelligence* 17(7):689–699.

Tsay, R. S. 1988. Outliers, level shifts, and variance changes in time series. *Journal of forecasting* 7(1):1–20.

Weber, B. G.; Mateas, M.; and Jhala, A. 2012. Learning from demonstration for goal-driven autonomy. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*.

Wilson, M. A.; McMahon, J.; and Aha, D. W. 2014. Bounded expectations for discrepancy detection in goal-driven autonomy. In *AI and Robotics: Papers from the AAAI Workshop*.

Automated Verification of Social Laws Robustness for Reactive Agents

Alexander Tuisov
Technion, Israel
alexandt@campus.technion.ac.il

Erez Karpas
Technion, Israel
karpase@technion.ac.il

Abstract

Coordinating agents in a multi-agent system is an interesting and important challenge. One of the most effective methods of coordinating multiple agents is using a "social law", which a-priori restricts some possible behaviors in order to ensure every agent can achieve its goal. Recent work has connected social laws with automated planning, and shown how to verify if a given social law is robust, that is, ensures each agent can achieve its goal regardless of the plans chosen by the other agents. This prior work assumed the agents choose a plan offline, and never modify it in response to the other agents' actions. In this paper, we address *reactive* agents, that is, agents that can reconsider their course of action during execution. This setting presents a new challenge, as agents now have the possibility of entering into an infinite loop (a livelock) in which each agent replans in the same way in response to the other agents. We show how to verify if a given social law is robust in such a setting, and specifically show how to verify that the social law is livelock-free via a compilation we call hindsight intent attribution.

Introduction

Systems with multiple autonomous agents are becoming more and more common (e.g., in robotic fulfillment centers) and will become more so in the future (e.g. autonomous cars, drone delivery). Designing such systems is very challenging; one of the main reasons for this is the need to coordinate all of the agents operating in the same shared environment.

Several approaches for coordination have been explored in the past. One possibility is to use a centralized controller, which controls the actions of all the agents (for example, see (Nissim and Brafman 2012)). However, this centralized control is not feasible for a large number of agents, especially when they are owned by different entities (as is the case for autonomous cars). Another approach is to allow each agent to act autonomously, and devise "rules of encounter" for when two agents come into conflict, which usually requires some negotiation between the agents (Georgeff 1988). In this paper, we follow a third approach, of enacting a "social law" (Shoham and Tennenholtz 1992) which restricts the behavior of the agents in order to ensure each agent can achieve its own goal. One of the main advantages social laws is they do not require any communication between the agents.

Previous work (Karpas *et al.* 2017; Nir and Karpas 2019) has shown how to verify if a given social law is *rationally robust*, that is, ensures that each agent can achieve its goal regardless of the (goal-achieving) plans chosen by the other agents. However, the kind of robustness described in these works is extremely strict — it requires every plan chosen *offline* by every agent to work, regardless of what every other agent is doing. Importantly, this assumes that agents are blind, in the sense that they never change their plan, regardless of what they see the other agents doing. This is an extremely strict requirement, and very few real-world problems have meaningful social laws that are *rationally robust*.

In this paper, we propose a new model, in which agents are *reactive*, that is, they are allowed to *replan* if they notice that their current plan is not going to work. We begin by formulating the execution and replanning model more precisely. We then propose a suitable definition of robustness for this model, and discuss the possible failure modes under this new model. Importantly, when agents are allowed to replan, a new failure mode called *livelock* emerges, in which two or more agents can get stuck in an infinite loop. Finally, we show how to verify the robustness of a given social law under our new model via heuristic search on a black-box planning model.

Background

In this section we provide an overview of concepts which provide a foundation for the rest of the paper.

Following Brafman *et al.* (2008) we define an MA-STRIPS problem as a quadruple $\Pi = \langle F, \{A_i\}_{i=1}^n, I, \{G_i\}_{i=1}^n \rangle$ where F is a set of predicates, A_i is a set of actions for agents numbered $1 \dots n$, $I \subset F$ is the initial state and G_i is the goal of agent i. Each action $a \in A_i$ can be described as a triplet $\langle pre(a), add(a), del(a) \rangle$, where $pre(a) \subset F$ is the precondition for performing a, and $add(a), del(a)$ are the add and delete effects of a respectively.

Social laws are a set of rules that regulate the behaviour of agents, such that a certain level of coordination is enforced upon the otherwise "selfish" agents. Such rules exist in human society (Rousseau 1762) as well as in artificial systems (Shoham and Tennenholtz 1995; Moses and Tennenholtz 1995; Agotnes *et al.* 2008). Social laws can be for-

malized in terms of MA-STRIPS tasks as a modification Π^l made to a MA-STRIPS task Π, in particular: facts, actions, preconditions and effects of actions, facts in initial and goal states added and/or removed from a problem. In the spirit of Karpas *et al.* (2017) we allow some preconditions to be marked as *waitfor*, and by doing so, assume our agents have the ability to *wait and do nothing*. Marking precondition as *waitfor* means that if an agent is due to perform an action a, and some $pre(a)$ is not fulfilled but is marked as *waitfor*, the agent will wait until $pre(a)$ becomes true, and then execute a.

Given a social law, one needs a way to assess how well it promotes the implicit coordination of agents. Karpas *et al.* (2017) propose to check a social law for *rational* and *adversarial* robustness. Enforcement of a rationally robust social law ensures that *every* agent may plan *offline* with no regard to plans and actions of other agents, and is still guaranteed to eventually reach its goal. Note that this requirement is very strict. In this paper we stick to robustness as a measure of the quality of a social law, but we attempt to derive a new, more liberal, notion of robustness.

The last tool we will require is the black-box planning formalism. In the spirit of (Frances *et al.* 2017), we define a *factored state model problem* (also known as *black-box planning problem*, in what follows we favor the use of the latter term) as a tuple $\Pi = \langle V, D, s_0, G, Act, A, t \rangle$ where:

- V is a set of variables X
- D_X are the domains of the variables $X \in V$
- s_0 is a full assignment to the variables in V that represents the initial state of the problem
- G is a conjunction of goal conditions
- Act is a set of actions (defined only as symbols)
- $A : S \mapsto 2^{|Act|}$ is a function that represents the set of actions applicable in each state
- $t : S \times A \mapsto S$ is a transition function

Note that we allow A and $t(s, a)$ to be given by black-box procedures rather than being defined explicitly. After having all the necessary tools covered, we can describe the exact model of the problem at hand.

Planning and Execution Model

We can now describe the planning and execution model for how the agents operate, which will allow us to define a new notion of robustness.

We will distinguish between a number of different models with regards to replanning while acting: First, and perhaps the most obvious setting is the one where replanning is forbidden altogether, i.e. every agent plans exactly once before the start of execution. This correlates to the basic model presented in (Karpas *et al.* 2017). The robustness of the social laws in this model had been already sufficiently explored there, and henceforth will be denoted as *rational robustness*.

Second, we will mention the setting where the agent replans after its every action. The robustness with respect to this model will be denoted as *anytime robustness*. Note that we do not put any limitations on the agent's plan, thus in many domains models of this kind will contain a trivial livelock, akin to the Buridan's ass dilemma.

The next type of model is on a spectrum between the options presented earlier ($never$ allow replanning and $always$ allow it), whereas here we allow the agent to replan only *on need*, defined as a state where the agent deduces that the original plan is no longer valid. Thus the need for replanning arises. Note that in the general case an agent may predict that its plan cannot succeed in advance. Replanning *on need* presents a wide range of possible rules for deducing where the need appears, and such an inference would probably require combining multi-agent reasoning and plan recognition about other agents' actions, which is potentially a hard task.

Thus, we would like to focus our discussion here on a specific sub-case of *replanning on need*: *reactive replanning*. Agents replan only when they cannot execute the next action in the original plan, i.e. the next action in the plan has an unfulfilled non-$waitfor$ precondition when the agent is activated by an external scheduler. This allows us to forgo the replanning in situations where the missing precondition of an un-executable action was restored by another agent. The robustness with respect to this model will be denoted as *reactive robustness*. Formally:

Definition 1. *A social law l for multi-agent setting $\Pi = \langle F, \{A_i\}_{i=1}^n, I, \{G_i\}_{i=1}^n \rangle$ is reactively robust iff: for all agents i, for all action sequences π which can result from initial planning, execution and reactive replanning, π achieves $G_1 \cup \cdots \cup G_n$*

Note that the definition requires *any* sequence to lead the agents to their goals, which means any proof of robustness will have to assume *adversarial* scheduler deciding which agent will act next.

Let us examine the relation between *rational* and *reactive* robustness. We will establish a hierarchy of robustness types by proving the following theorem:

Theorem 1. *For any problem Π and social law l, rational robustness is strictly stronger than reactive robustness.*

Proof. Rephrasing the theorem, every Π under l that is rationally robust is also reactively robust, but not vice versa. If Π under l is rationally robust, it is guaranteed that any set of plans the agents produce at the beginning will be executed to the end without a need to replan. Thus, replanning will never occur, and the initial plans will be executed until the goal is reached for every agent.

If, however, Π under l is reactively robust, we will show an example where it is not rationally robust. Consider the following setting: two agents a_1 and a_2, each has a goal to hammer its own nail (n_1 and n_2 respectively), and there are two distinct hammers (h_1 and h_2). In a setting where replanning is forbidden, an empty social law $\emptyset$ will not be *rationally* robust, since both agents can choose a plan in which they pick up h_1 and use it to hammer their nail, leading to failure. On the other hand, in a reactive setting an agent can replan to use h_2 instead, i.e. the empty social law is *reactively robust*. □

Having settled on an execution model, we need to explore in detail what can disrupt robustness. We now discuss the types of failure that could occur during execution.

Types of failure

Reactive robustness can be violated in a few ways:

- Deadend - an agent should act but does not have a defined action to execute for the current state, i.e. an agent cannot execute the next action from its current plan, and the replanning does not yield any other valid plan. Note that even if an agent i is stuck in a deadend, it is possible that in the future other agents' actions can make $i's$ plan for the goal possible again, but we still regard a possibility of a deadend as a violation to robustness.
- Deadlock - A state where no agent can perform any action, i.e. every agent is either waiting or finished. Since we allow $waitfor$ preconditions, some agents can be waiting. If every active agent waits for some $waitfor$ precondition to become true, a deadlock occurs.
- Livelock - a condition where one or more agents change the state of the system continuously, but no agent makes progress towards its goal. The oscillations of the system states continue *ad infinitum*, but the goal of some agents is never achieved.

Note that the types of failure in this work quite differ from those presented in (Karpas *et al.* 2017). A *rationally robust* system cannot enter a state of livelock, since no replanning occurs. On the other hand, in a reactive system an agent cannot fail because of a missing precondition, and will replan instead of declaring failure in that case.

As we have established the failures that lead to a breach in robustness, in the following section we propose a way to check whether a given social law is robust by actively searching for failures of a given types.

Reactive Robustness Verification

In this section we show a compilation for verifying *reactive robustness* of a MA-STRIPS problem Π under a social law l via black-box planning. Our strategy at the macro level is to map possible failures described in Section to the goals of the search problem we are constructing. The rest of the section is structured as follows: we first present the single agent projection of the task as a building block for the complete compilation, i.e. the planning task that every agent solves independently, and explain how the single agent plans are executed. We then show the complete compilation, using plans of individual agents as a building block.

Single Agent Projection

We first show the search space derived from the original MA-STRIPS problem for each individual agent. The main challenge is how to plan for each agent independently of the goals of the other agents, while still taking into account possible actions of the other agents.

In this work, we cannot use the single-agent projection used in (Karpas *et al.* 2017), since their single-agent projection does not allow to come up with plans that include an action that has a waitfor precondition on some fact which is false in the initial state, e.g.,, to wait for some other agent to move out of the way. In the reactive case it is even more important to allow waiting for other agents, since there are many more opportunities to plan for something which is currently false, but could become true later. Although there are works that try to emulate some aspects of reactive behavior in offline planning, these works operate under some very specific assumptions that do not apply to our task. For example, (Domshlak 2013) assumes there is a constant amount of "failure" outcomes, while in our setting the amount of plans an agent can come up with after replanning (and, by extension the amount of outcomes of any action that can invoke replanning) is far from constant. Thus, we propose the following single-agent projection of the problem:

Each agent i solves a classical planning task with a modified action space. The problem the agent solves includes only the actions of the agent itself (with a slight addition), the variables of the original problem, the initial state of the original problem and the private goal of the agent itself. The additional actions try to capture the effects of the environment and other agents that are out of i's control. To this end, for each action a_i of agent i we add a special action $waitfor\text{-}enable\text{-}a_i$, defined as:

- $pre(waitfor\text{-}enable\text{-}a_i) = pre(a_i) \setminus pre_w(a_i)$.
- $eff(waitfor\text{-}enable\text{-}a_i) = pre_w(a_i)$

where $pre_w(a_i)$ is the set of all *waitfor* preconditions of a_i. Such an action allows us to plan to wait for the *waitfor* preconditions of a_i. Note that no $waitfor\text{-}enable\text{-}a_i$ will ever have a *waitfor* precondition.

In conclusion, given an MA-STRIPS problem $\Pi = \langle F, \{A_i\}_{i=1}^n, I, \{G_i\}_{i=1}^n \rangle$, the single agent projection for agent i is the planning task $\Pi'_i = \langle F', A', I', G' \rangle$, where:

- $F' = F$
- $A' = A_i \cup \{waitfor\text{-}enable\text{-}a_i \mid a_i \in A_i\}$
- $I' = I$
- $G' = G_i$

After solving Π'_i, the execution policy of agent i can be extracted from the plan in a straightforward way:

- If the agent had achieved its goal: return a special action $finish{-}i$. Otherwise:
- If there is an action $a \in A_i$ planned for the next step and it can be executed: execute it.
- If there is an action $a \in A_i$ planned for the next step and it cannot be executed: replan. If replanning yields no solution, the next action is undefined for the current state, which constitutes a deadend.
- If there is an action $waitfor\text{-}enable\text{-}a_i$ planned for the next step: check if a_i can be executed. If so, execute a_i, otherwise, wait.

Complete Compilation

After presenting the planning and execution for each individual agent, we have the necessary tools to describe the complete compilation of reactive robustness verification to black-box planning. Consider a description of an MA-STRIPS task, a set of n agents, each agent i having its own plan, which is assumed to lead it to its goal (assuming the other agents do not interfere, and, in fact, help in achieving conditions specified by $waitfor\text{-}enable$ actions). We would like to check if the whole problem is *reactively robust*, which will be achieved by solving a *black-box planning* task, whose objective is to find a counterexample to

robustness, i.e. a failure. As was discussed in the previous section , a failure is either reaching a state of deadend, deadlock, or livelock.

The main driving logic behind this compilation is straightforward: we would like our planner to seek for a failure. As Definition 1 requires any sequence of actions to lead to the goal, the planner has the ability to choose which agent to activate next, and to control the result of each agent's replanning process when allowed. We now explain the state variables, actions, and the 3 different goal conditions.

State Variables: To make sure agents only replan when allowed (i.e., necessary) our compilation keeps track of both the state of the world and of each agent's current plan, which we refer to as the *internal state* (and is represented as the variables π_i in the compilation). Additionally, as we explain below, we keep another set of variables to detect potential livelocks, as well as other auxiliary variables.

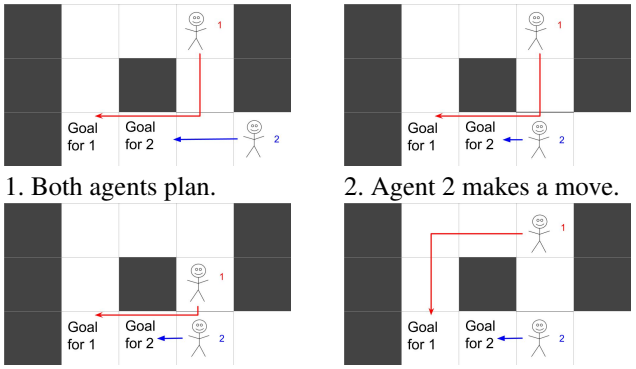
Note that unlike the compilation of (Karpas *et al.* 2017) we do not have a need to create a copy of the state for each agent. This is because we use the power of the black box successor generator to ensure that every possible plan leads to a goal in the single agent projection. Subsequently, we also do not need to create multiple versions of each action.

Actions: The actions that are available for the planner will be called $activate\text{-}action\text{-}a_i$, which have the following semantics: activate agent i and make it execute its next planned action a_i (that is determined according to i's current plan π_i). The effect of $activate{-}action{-}a_i$ on the state will be determined by the effect of a_i. If the next action cannot be executed, and the agent replans, the planner can execute one of the special actions $set\text{-}\pi_i\text{-}to\text{-}\pi$, which should set π_i to a certain plan π (out of every plan available to i). The $set\text{-}\pi_i\text{-}to\text{-}\pi$ actions can be though of as choosing the result of replanning. The amount of $set\text{-}\pi_i\text{-}to\text{-}\pi$ actions is enormous, so in the following section we describe a more compact compilation.

Goal (deadend): Detecting a deadend has been already discussed before. A deadend is a state where an agent has to replan, but its single agent projection is unsolvable. If such an agent exists in a state, that state is a goal state, and the complete compilation will recognize it by adding $failure$ in the transition to that state.

Goal (deadlock): Since the social law can declare some preconditions as *waitfor*, a deadlock where every agent waits for some *waitfor* precondition, and none of them can be activated, could occur. In a reactively robust environment deadlocks are not allowed, thus we give the planner an ability to detect them and declare failure (i.e., reach the goal) accordingly. A deadlock occurs at state s if for every agent i, the first action in $\pi(s)_i$ has some unfulfilled *waitfor* precondition or i has already finished (indicated by flag fin_i). We check this by raising a flag $i{-}is{-}waiting$ whenever agent i is waiting, and having an action $declare{-}deadlock$ with precondition $\forall i : i{-}is{-}waiting$, and an effect of achieving a failure.

Goal (livelock): A potential livelock can be formalized as the existence of an *infinite joint sequence of actions* that agents follow, never arriving at their intended goals. As shown by Patrizi *et al.* [2011] for planning with LTL goals,



1. Both agents plan. 2. Agent 2 makes a move. 3. Agent 1 comes to a point where it needs to replan. 4. Same world state as (2) with different internal states — no livelock.

Figure 1: Why Internal States Matter. Arrows represent plans.

infinite plans can be characterized as consisting of two sequences of actions: a sequence p_1 that maps the initial state of the problem to some state $\overline{s}$, and a second non-empty sequence of actions p_2 that maps $\overline{s}$ to itself and can be repeated indefinitely. Note that in order for a livelock to form, the whole state $\overline{s}$ should be repeated: both the world state and the agents' internal states, denoted by $\pi(\overline{s})_1 \dots \pi(\overline{s})_n$.

The need for replicating the internal states is best illustrated by an example as given in Figure 1. In this example, although the same world state is reached twice (with different internal states), there exists no livelock.

In the complete compilation, similarly to the compilation for LTL goals (Patrizi *et al.* 2011), we introduce a flag l and an action $start{-}loop$ that captures the transition between p_1 and p_2, as well as the auxiliary set of variables L, that stores the state $\overline{s}$ (of the original MA-STRIPS task and the agents' internal states) when action $start{-}loop$ is applied. A livelock is detected if the system can return to state $\overline{s}$ (via a non-empty sequence of actions). This is implemented by a goal test which checks whether $L = \overline{s}$ (to make sure this is not achieved by an empty sequence, we also use the flag $just{-}started{-}loop$). The correctness of this part of the compilation is shown by the following Lemma.

Lemma 1. *The state where $L = \overline{s}$ will be encountered if and only if there exists cycle p_2 that can be repeated infinitely many times.*

Proof. If $L = \overline{s}$ then the flag l was up, and the system returned to $\overline{s}$ after executing some non-empty set of actions p_2. We do not assume fairness on our scheduler, thus it can repeat the agent activation sequence that led to p_2. We also do not assume anything on our planners, thus they can generate identical plans from identical states, and that means that every action in p_2 can have the same effect as it had the first time. Starting from the same state, the same sequence of actions with the same results will lead to the same state again. □

We can now characterize the complete compilation, given an MA-STRIPS problem $\Pi = \langle F, \{A_i\}_{i=1}^n, I, \{G_i\}_{i=1}^n \rangle$, we construct the black-box problem $\Pi' = \langle V, D, s_0, G, Act, A, t \rangle$:

- $V = F \cup \{fin_i, state_i, i{-}is{-}waiting, \pi_i, i{-}replans | i = 1 \dots n\} \cup \{failure, l, just{-}started{-}loop\} \cup L$, where $L \equiv$

$F' \cup \{fin'_i, state'_i, i{-}is{-}waiting', \pi_i | i = 1 \dots n\}$, and π_i is the internal state of agent i

- $D_X = \forall X \in V \setminus IS_i : X \in \{true, false\}$. $\forall \pi_i : \pi_i \in \Pi_i^{all}$, where Π_i^{all} is the set of all possible plans for i
- $s_0 = \{f | f \in I\} \cup \{\neg f | f \in \overline{I}\} \cup \{\forall i : \neg fin_i, \neg i{-}is{-}waiting, \neg i{-}replans\} \cup \{\neg failure, \neg l, \neg just{-}started{-}loop\} \cup \{\forall x \in L : \neg x\} \cup \{\forall x \in \{\pi_i\} : x = \emptyset\}$
- $G = \{failure\}$
- $Act = \{activate{-}action{-}a_i | a_i \in A_i\} \cup \{set\text{-}\pi_i\text{-}to\text{-}\pi \mid \pi \in \Pi_i^{all}\} \cup \{start{-}loop, declare{-}deadlock, declare{-}livelock\}$
- A:
  - $pre(activate{-}action{-}a_i) = \{s \models pre_{waitfor}(a_i), \neg fin_i, \} \cup \{\forall i : \neg i{-}replans\}$.
  - $pre(declare{-}deadlock) = \{\forall i : i{-}is{-}waiting \cap fin_i\} \cup \{\exists i : \neg fin_i\}$
  - $pre(start{-}loop) = \emptyset$
  - $pre(declare{-}livelock) = \{l, \neg just{-}started{-}loop\} \cup \{L = s\}$
  - $pre(set\text{-}\pi_i\text{-}to\text{-}\pi) = i{-}replans$
- $t(s, a)$:
  - $del(just{-}started{-}loop) \in t(s, a)$ for every action except $start{-}loop$.
  - if a_i is defined and $a_i \neq finish{-}i, wait_i, replan$: $t(s, activate{-}action{-}a_i) = eff(a_i), del(i{-}is{-}waiting)$
  - else if $a_i = replan$: $t(s, activate{-}action{-}a_i) = add(i{-}replans)$
  - else if $a_i = wait_i$: $t(s, activate{-}action{-}a_i) = add(i{-}is{-}waiting)$
  - else if $a_i = finish{-}i$: $t(s, activate{-}action{-}a_i) = add(fin_i)$
  - else $t(s, activate{-}action{-}a_i) = add(failure)$
  - $t(s, set\text{-}\pi_i\text{-}to\text{-}\pi) = \pi_i \leftarrow \pi, del(i{-}replans)$
  - $t(s, start{-}loop) = add(l, just{-}started{-}loop), L \leftarrow s \setminus \{L, l, failure\}$
  - $t(s, declare{-}deadlock) = add(failure)$
  - $t(s, declare{-}livelock) = add(failure)$

The proof of correctness of the complete compilation follows the structure of the compilation itself, and is omitted for the sake of brevity. Note that the state space of the complete compilation is infinite (or at least doubly exponential if individuals plans are restricted to have no loops). In the following section, we present the *hindsight intent attribution* compilation, which eliminates the need to keep track of the internal states, at the cost of a more complex livelock goal test.

Hindsight Intent Attribution

As mentioned above, planning in the complete compilation is infeasible, due to the large branching factor (i.e., the number of possible single agent plans). We now propose the hindsight intent attribution compilation, which does not keep track of the internal states. Instead, the goal test for livelock looks at the plan so far, and attempts to attribute an intent (that is, an internal plan) to each relevant replanning decision of each agent in hindsight. Note that we need to keep track of when each agent replanned, but this can be done by adding one bit per agent for each state.

We must first modify the successor generator to only generate actions which could have been the next action now, in the plan that was generate when the agent last replanned. This is done by solving a planning problem similar to plan recognition as planning (Ramírez and Geffner 2009), where the actions that have been executed since the last replanning of agent i are "observed". To check if action a_i is a possible successor, we check whether there is a solution in which a_i is also "observed". As this could become expensive, we can also perform these checks only retroactively once a possible plan is found, and try to justify that plan.

To detect a livelock, when the system arrives at the same world state for the second time along a path, we declare a $pseudo - livelock$. Note that we also omit the L variables, as we can check whether we have reached the same state twice by tracing the parent pointers of the current node back up to the root. We then check whether this is a true $livelock$ by finding a plan for each relevant point along the current path where an agent replanned, which justifies the current path and could lead to a livelock.

In more detail, we will require auxiliary definitions of three distinct states for each agent: s_i^0, s'_i, s''_i, and one global state: s. These are defined as follows:

- s - a world state where $pseudo - livelock$ occurred. By definition of $pseudo - livelock$ it is a world state that occurred more than once during the execution. Therefore, we can divide the execution to periods according to the visits to s, i.e. refer to the time point of the last visit to the state s as t, and time point of a previous visit as $t - 1$ (note that we consider all previous visits to s).
- s_i^0 - the last state where agent i planned before $t - 1$. Can stem from either i replanning or i planning for the first time in the initial state.
- s'_i - the state where agent i *first* invoked replanning after $t - 1$ and before t.
- s''_i - the state where agent i *last* invoked replanning after $t - 1$ and before t. Can be the same as s'_i if i replanned only once between $t - 1$ and t.

Given a $pseudo - livelock$ happened at s, we need to know whether it is a real livelock. Livelock requires the agents to have the same internal state at t as in $t - 1$. We will denote the state of the complete compilation described in the previous section as $S = (s, \pi_1(s), \dots, \pi_n(s))$, where $\pi_i(s)$ are the agents' current plans. In a slight abuse of notation we will denote i's plan when it arrives to s at t by $\pi_i(s_t)$. Lemma 1 assures that livelock $\iff S_{t-1} = S_t$, thus it is sufficient to show that $(s, \pi_1(s_t), \dots, \pi_n(s_t)) = (s, \pi_1(s_{t-1}), \dots, \pi_n(s_{t-1}))$.

From here on, we look at each individual agent i and reason whether there is *a possibility* that $\pi_i(s_{t-1}) = \pi_i(s_t)$. For that, we divide the agents into three categories with regard to their behavior between s_{t-1} and s_t:

- Agents that did not perform any action between $t - 1$ and t – since there was no action performed by those agents, their internal state remains the same. These will be called *irrelevant agents*.
- Agents that performed some actions between $t - 1$ and t, but did not replan in this interval – their internal state *must* have changed, which means that if there is at least one agent in this category, s is not a true livelock. These will be called *advancing agents*.
- Agents that have both performed actions and replanned

between $t-1$ and t: for each such agent i we propose to create a planning problem that has a solution if and only if $\pi_i(s_{t-1})$ *could* have been equal to $\pi_i(s_t)$. These will be called *loop agents*.

Any loop agent arrives at s_{t-1} with the plan it conceived at s_i^0. The suffix of this plan is $\pi_i(s_{t-1})$. A second time around, it arrives at s_t with the plan it conceived at s_i'' and has $\pi_i(s_t)$ as a suffix. This means we have to check for the existence of two plans: one from s_i^0 with some prefix up to s (which we will call $\pi_i(s_i^0, s)$) and suffix $\pi_i(s_{t-1})$, and a second plan from s_i'' via s, whose prefix we denote by $\pi_i(s_i'', s)$, such that their suffixes match. Of course, we also require the plans to be consistent with the actions already observed (i.e., executed on the current path). Figure 2 provides graphical intuition.

For each agent i independently, given its single agent projection $\Pi_i' = \langle F', A', I', G' \rangle$ (described above) we will construct a classical planning problem $\Pi_i = \langle F_i, A_i, I_i, G_i \rangle$ that will have a solution iff a pair of plans as described above exists. First, we need both plans to have different prefixes and the same suffix. We split the plans into 4 phases: in phase 1 we follow the observed plan ($\pi_i(s_i^0, s)$) from s_i^0 to s, and in phase 2 we we follow the observed plan ($\pi_i(s_i'', s)$) from s_i'' to s. Then, in phase 3, the plans merge and reach state s_i' following the observed plan $\pi_i(s, s_i')$, and finally in phase 4, we diverge from the actions that have been observed, and simply need to find a plan that reaches the goal. To keep track of these separate plans, we create 2 copies of the state variables. Actions in phases 1 and 2 affect only one of these copies, while actions in phases 3 and 4 affect both copies. This is similar to the GRD compilation (Keren *et al.* 2014), except with merging instead of splitting. We denote the lengths $|\pi_i(s_i^0, s)|$ by l_1, $|\pi_i(s_i'', s)|$ by l_2, and $|\pi_i(s_i'', s)|$ by l_3, and define:

- $F_i = \{f_1, f_2 \mid f \in F'\} \cup \{fin\text{-}ph\text{-}k \mid k \in \{1,2,3\}\} \cup \{allow\text{-}a_j^{ph_k} \mid k \in \{1,2,3\}, j \in \{1, \ldots, l_k\}\}$
- $A_i = \{a_j^{ph_k} \mid a \in A', k \in \{1,2,3\}, j \in \{1, \ldots, l_k\}\} \cup \{a^{ph_4} \mid a \in A'\}\}$ where:
  - for $k \in \{1,2\}$:
    $pre(a_j^{ph_k}) = \{f_k \mid f \in \text{pre}(a)\} \cup \{allow\text{-}a_j^{ph_k}\}$,
    $add(a_j^{ph_k}) = \{f_k \mid f \in \text{add}(a)\} \cup \begin{cases} \{allow\text{-}a_{j+1}^{ph_k}\} & j < l_k \\ \{fin\text{-}ph\text{-}k, allow\text{-}a_1^{ph_{k+1}}\} & j = l_k \end{cases}$
    $del(a_j^{ph_k}) = \{f_k \mid f \in \text{del}(a)\} \cup \{allow\text{-}a_j^{ph_k}\}$,
  - $pre(a_j^{ph_3}) = \{f_1, f_2 \mid f \in \text{pre}(a)\} \cup \{allow\text{-}a_j^{ph_3}\}$,
    $add(a_j^{ph_3}) = \{f_1, f_2 \mid f \in \text{add}(a)\} \cup \begin{cases} \{allow\text{-}a_{j+1}^{ph_3}\} & j < l_3 \\ \{fin\text{-}ph\text{-}3\} & j = l_3 \end{cases}$
    $del(a_j^{ph_3}) = \{f_1, f_2 \mid f \in \text{del}(a)\} \cup \{allow\text{-}a_j^{ph_3}\}$
  - $pre(a^{ph_4}) = \{f_1, f_2 \mid f \in \text{pre}(a)\} \cup \{fin\text{-}ph\text{-}3\}$
    $add(a^{ph_4}) = \{f_1, f_2 \mid f \in \text{add}(a)\}$
    $del(a^{ph_4}) = \{f_1, f_2 \mid f \in \text{del}(a)\}$
- $I_i = \{f_1 \mid f \in s_i^0\} \cup \{f_2 \mid f \in s_i''\} \cup \{allow\text{-}a_1^{ph_1}\}$
- $G_i = \{f_1, f_2 \mid f \in G'\} \cup \{fin\text{-}ph\text{-}3\}$

To prove the correctness of this compilation, note that the existence of this pair of plans (e.g., a solution to Π_i) indicates that it is possible that $\pi_i(s_{t-1}) = \pi_i(s_t)$ if the agent was acting alone. Moreover, by definition of s_i'' we know that agent i did not replan from s_i'' to s, which means $\pi_i(s_i'', s)$ can be executed regardless of the plans of the other agents. This decoupling gives us the ability to reason about each agent independently, as we already know how these plans should be interleaved to achieve the loop we have already observed. Thus, to reach the conclusion $S_t = S_{t-1}$ it is sufficient to: a) find such a pair of plans for each loop agent and b) show there are no advancing agents.

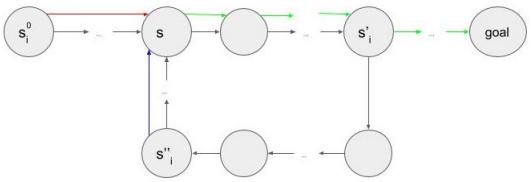


Figure 2: Graphical representation of the livelock loop from i's point of view. $\pi_i(s_i^0, s)$ is in red, $\pi_i(s_i'', s)$ is in blue, common suffix $\pi_i(s_t) = \pi_i(s_{t-1})$ is in green. Observed actions are in gray.

Theorem 2. *Hindsight intent attribution returns "true" for a state s where pseudo-livelock occurred if and only if a true livelock is possible from s.*

Proof. If a true livelock is possible from s, it implies that there exists a schedule of actions such that $(s, \pi_1(s_t), \ldots, \pi_n(s_t)) = (s, \pi_1(s_{t-1}), \ldots, \pi_n(s_{t-1}))$. This in turn implies that $\forall i : i \notin$ *advancing agents*. Also, for each $i \in$ *loop agents*, there exist two plans: π_i^1 planned before $t-1$ with suffix $\pi_i(s_{t-1})$, and π_i^2 planned after $t-1$ and before t, with suffix $\pi_i(s_t) = \pi_i(s_{t-1})$. Moreover, prefixes of those plans have been already executed, thus are compatible with observed actions. Thus, $\forall i : \pi_i^1, \pi_i^2$ is pair of plans that compose a solution for the hindsight intent attribution search procedure in a following way: from the points of the last replans before $t-1$ and t, the actions observed will be executed in their respective copy of facts, until both copies arrive to s. From s, $merge$ will be executed, and after that, merged version of $\pi_i(s_t)$ will complete the solution. We have shown a possible solution for each agent, therefore hindsight intent attribution would return "true".

Proving the other side of *if and only if*, assume hindsight intent attribution returns "true". It means, $\forall i \in agents : i \notin$ *advancing agents*. For each $i \in$ *irrelevant agents* trivially $\pi_i(s_t) = \pi_i(s_{t-1})$. For each $i \in$ *loop agents* there exist two plans found by the search procedure: $\pi_i(s_i^0, s) \cdot \pi_i(s_{t-1})$ and $\pi_i(s_i'', s) \cdot \pi_i(s_t)$, where $\cdot$ denotes concatenation. Moreover, these plans are consistent with the observed actions, and $\pi_i(s_t) = \pi_i(s_{t-1})$ by the correctness of the search procedure. This, in turn, means that each loop agent *could* have chosen plan $\pi_i(s_i^0, s) \cdot \pi_i(s_{t-1})$ in s_i^0, and plan $\pi_i(s_i'', s) \cdot \pi_i(s_t)$ from s_i'', and still remain consistent with the actions observed, independently of other agents' plans. The independence comes from the fact that both $\pi_i(s_i^0, s)$ and $\pi_i(s_i'', s)$ were executed fully without replanning, i.e. there exists a schedule such that $\forall a \in \pi_i(s_i^0, s), \pi_i(s_i'', s)$, $pre(a)$ are fulfilled regardless of plans of other agents. Thus, there exists a schedule s.t. $(s, \pi_1(s_t), \ldots, \pi_n(s_t)) = (s, \pi_1(s_{t-1}), \ldots, \pi_n(s_{t-1}))$, which means true livelock is possible from s. □

Conclusion

We have described an execution model which allows agents to adjust their plans *online* via replanning. We then proposed the notion of reactive robustness, which is less strict than rational robustness (Karpas *et al.* 2017; Nir and Karpas 2019). Finally, we have shown how to check whether a problem is robust via a compilation to black-box planning.

References

Thomas Agotnes, Wiebe van der Hoek, and Michael Wooldridge. Robust normative systems. In *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 2*, AAMAS '08, pages 747–754, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

Ronen I Brafman and Carmel Domshlak. From one to many: Planning for loosely coupled multi-agent systems. In *ICAPS*, pages 28–35, 2008.

Carmel Domshlak. Fault tolerant planning: Complexity and compilation. In *ICAPS*, 2013.

Guillem Frances, Miquel Ramírez Jávega, Nir Lipovetzky, and Hector Geffner. Purely declarative action descriptions are overrated: classical planning with simulators. In *IJCAI 2017. Twenty-Sixth International Joint Conference on Artificial Intelligence; 2017 Aug 19-25; Melbourne, Australia.[California]: IJCAI; 2017. p. 4294-301.* International Joint Conferences on Artificial Intelligence Organization (IJCAI), 2017.

Michael Georgeff. Communication and interaction in multi-agent planning. In *Readings in distributed artificial intelligence*, pages 200–204. Elsevier, 1988.

Erez Karpas, Alexander Shleyfman, and Moshe Tennenholtz. Automated verification of social law robustness in strips. *Distributed and Multi-Agent Planning (DMAP-16)*, page 73, 2017.

Sarah Keren, Avigdor Gal, and Erez Karpas. Goal recognition design. In *ICAPS*, 2014.

Yoram Moses and Moshe Tennenholtz. Artificial social systems. *Computers and Artificial Intelligence*, 14:533–562, 1995.

Ronen Nir and Erez Karpas. Automated verification of social laws for continuous time multi-robot systems. In *AAAI*, 2019.

Raz Nissim and Ronen I Brafman. Multi-agent a* for parallel and distributed systems. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 3*, pages 1265–1266. International Foundation for Autonomous Agents and Multiagent Systems, 2012.

Fabio Patrizi, Nir Lipovetzky, Giuseppe De Giacomo, and Hector Geffner. Computing infinite plans for ltl goals using a classical planner. In *IJCAI*, pages 2003–2008, 2011.

Miquel Ramírez and Hector Geffner. Plan recognition as planning. In *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

Jean-Jacques Rousseau. The social contract, 1762, 1762.

Yoav Shoham and Moshe Tennenholtz. On the synthesis of useful social laws for artificial agent societies (preliminary report). In *AAAI*, pages 276–281, 1992.

Yoav Shoham and Moshe Tennenholtz. On social laws for artificial agent societies: off-line design. *Artificial intelligence*, 73(1-2):231–252, 1995.