Goal Lifecycle Networks for Robotics

Mark Roberts[1] **Laura M. Hiatt**[1] **Vivint Shetty**[2]
Benjamin Brumback[1] **Brandon Enochs**[3] **Piyabutra Jampathom**[2]
The U.S. Naval Research Laboratory
[1]Code 5514 | [2]Code 8245 | [3]Code 5545
Washington, DC, USA | {first.last}@nrl.navy.mil

Abstract

A Goal Lifecycle Network (GLN) is a conceptual process model that captures the progression of goals from their formulation to their completion, including planning and execution concerns. GLNs synthesize the literature on hierarchical goal networks, goal lifecycles, and plan execution. We formalize GLNs based on a state-variable representation, extend GLNs with an execution lifecycle, describe a partial reference implementation of GLNs, and show how the temporal PDDL language can be translated into GLNs for dispatchable execution. We integrate GLNs in three proof-of-concept robotics demonstrations: (1) a two-armed robot sorting items into baskets; (2) a multi-vehicle quad-rotor team surveying a region; and (3) centralized planning for a simulated disaster relief based on the Robocup Rescue League. The theory, implementation, and demonstrations highlight that GLNs are effective for goal management in the robotics systems we study.

1 Motivation and Contributions

Goals are a characteristic of intelligent behaviour (e.g., (Hawes 2011; Aha 2018)) and goal lifecycles are often used to manage them in agents. Goal lifecycles have been applied to agent-oriented systems (Thangarajah et al. 2010; Harland et al. 2014) as well as robotic systems (e.g., (Roberts et al. 2016; Niemueller, Hofmann, and Lakemeyer 2019)) and cognitive systems (e.g., (Cox, Dannenhauer, and Kondrakunta 2017)). This family of lifecycles complement each other, each providing perspective on goal management.

We present a Goal Lifecycle Network (GLN) that synthesizes three lines of research by simplifying an existing goal lifecycle, linking it to a PDDL planner, and demonstrating its use in robotics. Our contributions include: *(1)* formalizing GLNs using a state variable representation (Ghallab, Nau, and Traverso 2016) with a simplified variant of the goal lifecycle by Roberts et al. (2016) and goal networks by Shivashankar et al. (2012; 2013); *(2)* extending GLNs with a simplified Execution Lifecycle; *(3)* revising work by Veloso, Pérez, and Carbonell (1990) to convert a temporal PDDL plan into a dispatchable GLN; *(4)* providing a GLN reference implementation, called ACTORSIM[1], that includes cognitive architecture elements; and *(5)* applying ACTORSIM and GLNs to three proof-of-concept robotics demonstrations.

[1]Please email primary author for latest code.

2 Preliminaries

GLNs synthesize a considerable literature, some of which will be new to most readers. So we review the state-variable representation (Section 2.1), Hierarchical Goal Networks (Section 2.2), and the goal lifecycle we extend (Section 2.3).

2.1 State-Variable (SV) Representation

For representing GLNs, we will adapt the notation of Ghallab, Nau, and Traverso (2016) and ANML as implemented in FAPE (Dvorak et al. 2014). PDDL is a common language for automated planning (Fox and Long 2003). Figure 1 shows the first few lines of a PDDL domain for the CrazySwarm Demo in Section 5. In this PDDL, vehicles traverse waypoints to take observations. A vehicle can have a status of landed, hovering, flying, or sensing. Several boolean predicates capture the state of objects in the world. Finally, actions are the way that agents modify the world; the example shows an action template (operator) for navigation. We will use this example in discussing the state-variable representation.

A *domain* is a triple $D = (S, A, \gamma)$, where S is a finite set of states (i.e., state variables), A is a finite set of actions, which are composed of statements, and $\gamma : S \times A \rightarrow S$ is a partial transition function.

States are composed of objects with their attributes and relations. Object instances can have a type, such as `v0` is a VEHICLE and `w1` or `w2` are WAYPOINTs. Objects can only be assigned or bound to an appropriate variable type.

A state relation (SR) relates objects and is defined by a state relation template (SRT), e.g., `can-traverse`(WAYPOINT, WAYPOINT). An SR instance binds objects to its parameters, e.g., `can-traverse`(`w1`, `w2`).

State variables are central to this representation because they are used to define goals in the form of statements. A state variable (SV) defines attribute of an object and its allowed values. An SV has variables that are parameters and variables that are values. For clarity, we will list variable parameters first and underline their values. An SV is defined with a state variable template (SVT) over types, such as `at`(VEHICLE, WAYPOINT). An SV instance binds objects to its parameter(s), such as sv_1=`at`(`v0`, WAYPOINT), where sv_1 labels the SV.

Returning to the example in Figure 1, it is straightforward to translate the predicates of temporal PDDL to SRTs and SVTs. Any single argument predicate is converted to an SVT with a boolean value. Multi-valued predicates convert to

```
(define (domain vehicle) (:requirements :typing :durative-actions)
  (:types vehicle waypoint stat)
  (:constants LANDED HOVER FLYING SENSING - stat )
  (:predicates (can_traverse ?x - waypoint ?y - waypoint)
               (inspected ?w - waypoint)
               (phenomenon_at ?w - waypoint)
               (at ?v - vehicle ?value-w - waypoint)
               (status ?v - vehicle ?value-s - stat) )
  (:durative-action navigate
    :parameters (?v - vehicle ?fr - waypoint ?to - waypoint)
    :duration (= ?duration 1)
    :condition (and (over all (can_traverse ?fr ?to))
               (at start (at ?v ?fr))
               (at start (status ?v HOVER)) )
    :effect (and (at start (not (at ?v ?fr)))
          (at start (not (status ?v HOVER)))
          (at start (status ?v FLYING))
          (at end (not (status ?v FLYING)))
          (at end (status ?v HOVER))
          (at end (at ?v ?to)) ) ) ... )
```

Figure 1: The beginning of the Temporal PDDL domain for the CrazySwarm demo in Section 5.

SRTs unless their last variable starts with *?value-*, in which case it is converted to an SVT. Let BOOLEAN$= \{true, false\}$ and STAT be the constant symbol in Figure 1. The only SRT of the domain is `can_traverse`(WAYPOINT, WAYPOINT) and the SVTs are: `inspected`(WAYPOINT, BOOLEAN), `phenomenon_at`(WAYPOINT, BOOLEAN), `at`(VEHICLE, WAYPOINT), and `status`(VEHICLE, STAT). We manually provide the *?value-* annotations for our demos, but it should be easy to infer these automatically for many domains.

A *statement* wraps an SV with an assigned value and temporal extent. A temporal interval $[t_1, t_2]sv_1$ indicates that sv_1 holds over $t_1 \leq t < t_2$ and the shorthand $[t_3]sv_1$ indicates that sv_1 holds for a single time point t_3. There are three *base statements*. Let $sv_1 =$ `at(v0,` WAYPOINT). A **persistence statement** $[t_{start}, t_{end}]sv_i ==$ `value` indicates a value remains constant over its interval. This is sometimes referred to as a temporal assertion. For example, $sv_1 ==$ `w0` says that `v0` stays at `w0` over the interval. A **transition statement** $[t_{start}, t_{end}]sv_i =$ `from` $\rightarrow$ `to` indicates a change of value over the interval. This is sometimes referred to as a change assertion. For example, $sv_1 =$ `w0` $\rightarrow$ `w1` says that `v0` starts at `w0` and transitions to `w1` at some point during the interval. An **assignment statement** $[t_{start}, t_{end}]sv_i :=$ `value` *sets* the value of sv_i at the start of its interval and remains true during the interval. It is useful for establishing the initial conditions of a situation. For example, $sv_1 :=$ `w0` sets the value of sv_1 to `w0` at t_1 and this value does not change for the interval. Additional statements may be created from these base statements. For example, the EXECUTIONSTATEMENT in Section 3.3 is a variety of transition statement. In Section 4, a **negated persistence statement** $[t_{start}, t_{end}]sv_i \neq$ `value` indicates sv_i must not equal value over its interval.

Statements are used within *actions* to denote its preconditions and effects. To illustrate, the $navigate$ operator from Figure 1 would be translated as follows. For easier reading of this example, we "instantiate" variables with placeholder instances; that is *?v* will be written `v`. Let $[t_{start}, t_{end}]$ be the interval of the entire action. Its conditions and effects are:

$[t_{start}, t_{end}]$ `can_traverse(fr, to)`
$[t_{start}]$ `at(v)` == `fr`
$[t_{start}]$ `status(v)` == `HOVER`
$[t_{start}, t_{end}]$ `at(v)` = `fr` $\rightarrow$ `to`
$[t_{start}]$ `status(v)` :=`FLYING`
$[t_{end}]$ `status(v)` :=`HOVER`

2.2 Simple Goal Networks

Hierarchical Goal Networks provide a way to order and link goals together. A goal network is solved when it is empty; i.e., there are no more goals to pursue. We will adapt the original definition of the Hierarchical Goal Network from Shivashankar et al. (2013), although here we call it a Simple Goal Network (SGN) and introduce it informally to mention its salient points. As defined in that original work,

Definition 1. A *simple-goal-network* $sgn = (G, \prec)$ is a pair where G is a set of nodes, each $g \in G$ is a predicate statement of ground literals in disjunctive normal form, and $\prec$ describes a partial order over G.

A problem for an sgn is a triple $P_{sgn} = (D_{\text{STRIPS}}, s_0, sgn_0)$ where D_{STRIPS} is a domain, s_0 is the initial state (i.e., a conjunction of ground literals), and sgn_0 is a *simple-goal-network*. D_{STRIPS} follows from the standard PDDL definition.

Several definitions apply to nodes of an sgn, as adapted from Alford et al. (2016), that help define the set of solutions. A node $g \in G$ is *unconstrained* if it has no predecessors; that is, $\forall_{g' \in G} g' \not\prec g$. The operation *release* removes an unconstrained node $g \in G$ from sgn, yielding sgn'; that is $sgn' = (G \setminus g, (g_1, g_2) \in \prec | g_1 \neq g)$.

A solution for P_{sgn} is defined recursively: (1) an empty network (i.e., $G = \emptyset$) is a solution; (2) if a state can entail any unconstrained node $g \in G$ then any solution for P' where g can be released is a solution for P; (3) if an action a is applicable to s_0, resulting in state $s' = apply(a, s_0)$, and some plan π can reach s', then $a \circ \pi$ is a solution.

Case (3) above means that classical planners can solve these problems by producing π for any valid g. Further, methods such as landmark extraction can identify which $g \in G$ is best to solve next.

Adding hierarchy is accomplished using methods. A *method* decomposes a goal g by inserting its subgoals into T and ordering them to occur before g. A method m is composed of a head $head(m)$, preconditions $pre(m)$, and a goal network m_{sgn}. The head consists of the name and parameters of the method, and the preconditions determine if the method is applicable. When applied, m_{sgn} is prepended into the existing network. Methods allow a domain designer to write domain-specific decompositions for goals. Shivashankar et al. (2013) showed that methods substantially improve search, when they are available. In the absence of methods, however, standard classical planning techniques still apply because classical planners work on goals

2.3 The Goal Node and an Existing Goal Lifecycle

The *simple-goal-network* provides a convenient way to order goals in disjunctive normal form. But a robotic system often tracks more detail about a goal than just the objective. At a minimum, the execution status of a goal is usually tracked. A goal lifecycle provides a way to track goals as they progress through a system.

Roberts et al. (2016) extended the goal node with a goal lifecycle. Key contributions of that 2016 lifecycle included the goal node, goal mode, and goal strategies. The *goal node* is a data structure that holds the history and future commitments for a goal. A *goal mode* is the "state" of the goal, which is progressed using a *goal strategy*. Goal strategies are a special kind of method (cf. Section 2.2) that progress a goal network. For example, a goal is created using the *formulate* strategy, which results in the *formulated* mode. On their way to being *finished*, goals progress through *selected*, *expanded*, *committed*, *dispatched*, and *evaluated*.

Figure 2 shows a variant of that lifecycle, which we simplified in three ways: (1) We remove task networks, which focuses the problems on goals. (2) We commit to a state variable representation, as opposed to a generic description provided in the 2016 discussion. (3) We show only the transitions that are actually used for this study.

3 New Goal Networks

We are now in a position to introduce the core contributions of this paper. We first synthesize simple goal networks with the SV representation into an extended goal network (Section 3.1). Then, we introduce the GLN (Section 3.2). Finally, we extend the GLN with an EXECUTIONSTATEMENT (Section 3.3).

3.1 Extended Goal Network (GN)

We can now update Definition 1 to use SV representations with the more general goal node.

Definition 2. An *Extended Goal Network* $gn = (\check{\mathrm{G}}, \prec)$ is a pair where $\check{\mathrm{G}}$ is a set of goal nodes, $\prec$ describes a partial order over $\check{\mathrm{G}}$, and each *goal node* $\check{\mathrm{g}} \in \check{\mathrm{G}}$ is a node containing a *statement*, as defined above. We will refer to this class of networks as GNs and specific network instances as a gn.

The problem statement and solutions to the GN are the same as for the *simple-goal-network*. The GN provides several innovations over Definition 1. Firstly, instead of a predicate in disjunctive normal form, the use of a statement restricts a goal node to a single temporal assertion or change assertion, simplifying how nodes can relate and making it easier to track execution. Secondly, the inclusion of a temporal interval will eventually allow a GN to incorporate more sophisticated temporal relationships beyond $\prec$; for example, future work can extend these temporal intervals to represent conjunctions and disjunctions. These improvements make the GN much easier to use for goal management and easier to integrate with the goal lifecycle, described next.

3.2 Goal Lifecycle Networks (GLN)

Goal Lifecycle Networks extend GNs with the goal lifecycle from Section 2.3. (i.e., MODES and STRATEGIES). We only need to update Definition 2 with a modest extension to incorporate the goal lifecycle.

Definition 3. A *Goal Lifecycle Network* $gln = (\check{\mathrm{G}}, \prec)$ is a pair where $\check{\mathrm{G}}$ is a set of goal nodes, $\prec$ describes a partial order over $\check{\mathrm{G}}$, and each *goal node* $\check{\mathrm{g}} \in \check{\mathrm{G}}$ contains a *statement* as well as a MODE. We will refer to this class of networks as GLNs and specific networks instances as a gln.

A GLN is a specialization of a GN that further restricts possible transitions via the goal lifecycle. A problem for a GLN is a tuple $P_{gln} = (D, \mathcal{N}, R)$ where D is an SV domain, $\mathcal{N} = (s_0, gln_0)$ is a pair of the starting state s_0 and initial GLN gln_0, and R is a set of refinement strategies for progressing goals shown in Figure 2. Similar to a *simple-goal-network*, a solution for a GLN is a sequence of refinement strategies $(r_1, .., r_n)$ where $gln_n = \emptyset$. In other words, the sequence releases goals until no goals remain. (Maintenance goals do not result in an empty network, which will be addressed in future work.)

A GLN's transitions for $\check{\mathrm{g}}$ are more restricted due to R, which we denote as STRATEGIES. Applying a strategy progresses a goal through the lifecycle. Strategies are adapted from the lifecycle by Niemueller et al. (2019), which is a variant of the lifecycle by Roberts et al. (2016).

3.3 The EXECUTIONSTATEMENT

To track the execution state of a goal node, the EXECUTIONSTATEMENT is a transition statement $[t_{start}, t_{end}]executed(a) == \text{INACTIVE} \rightarrow \text{COMPLETED}$ where: $executed$ is an SV, $a \in A$ is an action, and the values of $executed$ are EXECUTION-STATES shown in Figure 2. Additionally, the OUTCOMES can be one of SUCCESS, INTERRUPTED, or FAILED. Each Executive in Section 5 uses a domain-specific mechanism update the execution state and outcome. One or more EVALUATE strategies process these updates, eventually resulting in a FINISHED goal.

4 Creating Partially Ordered Plans

Many PDDL planning systems produce plans that can be more flexible with post processing. We adapt an algorithm due to Veloso et al. (1990) to relax PDDL plans that fosters effective, parallel execution of plans. The algorithm relaxes a totally ordered plan to a partial order plan by analyzing dependencies among the plan steps. The algorithm adds ordering constraints for: (1) an action a_i from each of the closest, prior action adding the preconditions of a_i, (2) an action a_i from each of the earlier actions requiring a_i's delete effects, and (3) an action a_i from each of the earlier actions deleting a condition added by a_i. Finally, transitive edges are pruned.

We modified this algorithm to accommodate our use of statements instead of the prior formulation of preconditions and effects. The original algorithm assumes that effects are added and deleted after an operator is executed. Statements hold over more flexible temporal intervals. Therefore, we modified the algorithm to account for positive or negative statements that can reconcile at any given time. For example, in Step 1, we expand the reasoning to account for how the operators added for sequencing with a_i could add the effect supporting the preconditions of a_i, either at the start or finish of its execution. We changed Steps 2 and 3 similarly.

5 Demonstrations using ACTORSIM

We describe three robotics demonstrations that use GLNs in varying ways[2]. For the first two systems, COVID-19 restric-

[2]Videos are available at: http://makro.ink/actorsim/

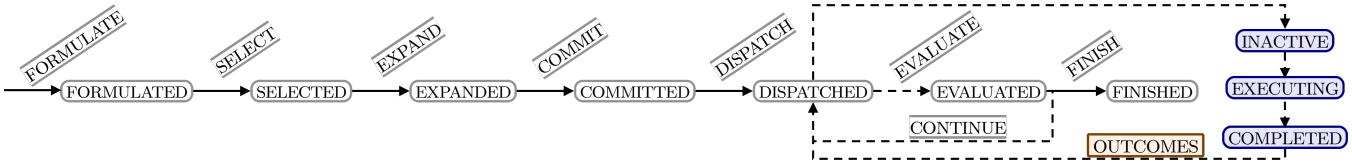

Figure 2: The simplified goal lifecycle. Goals progress via STRATEGIES into MODES. The dashed lines show how the Executive tracks and updates EXECUTION-STATE and OUTCOMES, which are managed by one or more EVALUATE strategies.

tions prohibited us from accessing our physical platforms, so we discuss simulations instead.

Overview of the Cognitive Architecture ACTORSIM uses a Cognitive Architecture for its agents. Cognitive architectures provide a way to characterize the design of an agent (Laird, Lebiere, and Rosenbloom 2017). Figure 3 shows an abstraction of ACTORSIM and how the contributions of this paper fit together. The central process in ACTORSIM is a CognitiveCycle that loops through four steps:

(1) Update Working Memory When the agent perceives, the CognitiveCycle converts these perceptions and updates the FactMemory, which holds objects and statements that are true about the world; cf. Section 2.1.

(2) Determine Applicable Strategies The StrategyMemory holds the goal strategies that progress goals in the GoalMemory; cf. Section 3.2. The CognitiveCycle determines which goal strategies in StrategyMemory are applicable.

(3) Rank Applicable Strategies The CognitiveCycle ranks applicable goal strategies using goal priorities and goal orderings held in the root of the goal network in $\prec$.

(4) Apply Applicable Strategies The Cognitive Cycle applies goal strategies from Figure 2, which progress a goal through the system. For example, the EXPAND strategy either decomposes a goal into subgoals or calls a temporal PDDL planner to create a plan for one or more goals. The COMMIT strategy converts a plan's steps into EXECUTIONSTATEMENTS, cf. Section 3.3, which track a goal during execution. For executives that can execute parallel plans, we convert the plan into a partial order plan; cf. Section 4.

In an independent process, the Executive reads these DISPATCHED EXECUTIONSTATEMENTS from the GoalMemory, executes them on the platform, and updates state in the WorkingMemory for the next cycle.

Controlling a two-armed robot We use GLNS to control a two-armed robot that manually sorts cans into different bins. The robot platform we use is the DRC-Hubo humanoid robot[3]. The robot has two 7-DOF arms that are equipped with a 1-DOF gripper. The robot has additional degrees of freedom in its waist and legs, which we do not use here.

We developed a temporal PDDL model for the robot for this sorting task. Temporal PDDL planning is done via TFD[4] (Eyerich, Mattmüller, and Röger 2009). Motion plans are computed using the Open Motion planning Library (OMPL)[5] (Sucan, Moll, and Kavraki 2012). Target positions are passed to the motion planner from the executive. These positions are converted to a set of joint angles for the Hubo's arm using inverse kinematics. Using the set of solved joint angles of the target positions, OMPL finds a path from the current position to the end position, while abiding by any required constraints, for example staying level with the table when pushing an object. This path of joint angles is then returned to the Hubo's hardware interface, or the Gazebo simulator[6], which executes the sequence of steps.

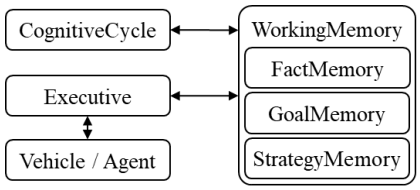

Figure 3: Overview of the key components in ACTORSIM.

The architecture controls the Hubo by connecting ACTORSIM and GLNS with the robot executive, motion planner and hardware. The main control loop is the executive, which that executes DISPATCHED nodes in the goal memory. The Hubo in the Gazebo simulator sorts four different cans into two different bins. Because of the spatial nature of the task, some actions can be executed concurrently without interfering with others (such as pre-positioning the hands for pickup). Other actions can only be done one at a time (such as an arm pushing a can into a basket on the other side of the table). We simplify perception by providing the robot with the positions of the cans directly.

At the start of the demonstration, a FORMULATE strategy creates goals to move each can to the basket with the matching color, and then have the two arms move back to their home position. EXPAND creates a TFD plan to inspect the phenomena, and translates it to a partial-order representation as described in Section 4. COMMIT converts this plan into EXECUTIONSTATEMENTS, while DISPATCH then sends one at a time to the Hubo executive described above.

Controlling multiple quadrotor systems We connected ACTORSIM to the CrazySwarm quadrotor environment (Preiss et al. 2017), which is an environment for controlling teams of micro-quadrotor systems. Each vehicle can accept commands to takeoff, land, and move to specific locations. Although the vehicles can move in continuous space, we command them to move along grid locations.

We developed a temporal PDDL model for the CrazySwarm environment, the first part of which is shown

[3] http://www.rainbow-robotics.com

[4] http://gki.informatik.uni-freiburg.de/tools/tfd/

[5] https://ompl.kavrakilab.org

[6] http://gazebosim.org

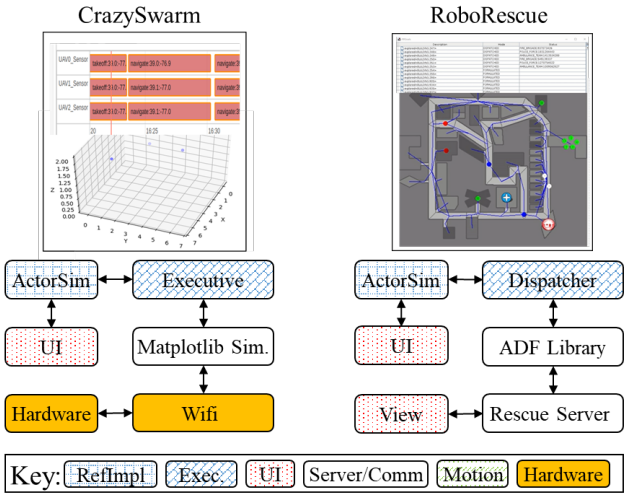

Figure 4: Video snaps and architectures for our demonstration: CrazySwarm (left) and RoboRescue (right).

in Figure 1. The model has four actions corresponding to the commands that be given to the vehicles. This model is converted into the SV representation and into GLNs.

We simulate three vehicles moving to different cells to inspect phenomena. Figure 4 (left) shows the control architecture. A Gantt-like User Interface (UI) displays the dynamic execution of goals; it connects to ACTORSIM through a javascript-xml bridge called JMX. This web-based UI adds goals as they are dispatched and provides updates as the goals complete. The executive reads any dispatched EXECUTIONSTATEMENTs from the goal memory and submits them to the simulator. When physical quadrotors are attached, these are also forwarded via WiFi to physical vehicles.

Vehicles start at the (x, y) origin $(0, 0)$ and inspect phenomena at three points: $(0, 5)$, $(2, 2)$, and $(5, 0)$. The UI translates these latitude-longitude coordinates to display the goals. A FORMULATE strategy creates goals to inspect the phenomena. During the EXPAND strategy, the POPF2 planner[7] (Coles et al. 2011) creates a centralized plan for the vehicles. COMMIT converts this plan into EXECUTIONSTATEMENTs, while DISPATCH then sends one at a time to the vehicle executive. A `matplotlib` visualization is shown of the vehicles executing their plans. As the vehicles move to the inspection points, goal status appears in the UI.

Controlling Teams for Disaster Relief The RoboCup Rescue Agent Simulator, or `Roborescue`, models a situation immediately following a natural disaster (Sheh, Schwertfeger, and Visser 2016). It is the basis of the RoboCup Rescue Simulation League (Akin et al. 2012). We simulate six agents helping civilians (green dots) reach a refuge (bottom-right building). Figure 4 (right) shows a screenshot from our video and the control architecture. Civilians will move along roads (light gray) if they are able but may become trapped under or behind rubble (solid black). If a civilian is trapped for too long or suffers too much damage then they turn black, indicating that they have died.

Controllable agents consist of police (blue dots), ambulance (white dots), and fire brigades (red dots). All three agents can $explore$ an road or building. Only police can $clear$ rubble to $unblock$ a human. Ambulances can $unbury$ any human and can $rescue$ a single civilian, which is to transport a civilian too damaged to walk on its own. Fire patrols can $douse$ fires using water from hydrants.

There are 71 STRATEGIES used in this demo. These are composed of ten strategies for seven goal types. A goal of $explored$(BUILDING) can be performed by any agent. Ambulance goals are $unburied$(HUMAN) and $rescued$(CIVILIAN), brigades can $douse$(BUILDING), and police can $unblock$(AGENT), $unblock$(CIVILIAN), or $clear$(RUBBLE). The ten strategies for each of these goals include: FORMULATE when appropriate, SELECT when an agent is available, SELECT by preempting an agent due to a goal ordering, always EXPAND because agents do their route planning, always COMMIT because agents do their route planning, DISPATCH the goal for execution, EVALUATE a goal as new information arrives, EVALUATE a preempted goal to place it back in FORMULATED, FINISH a goal that is COMPLETED, and DROP a goal that is FINISHED.

There are also three goal orderings: Ambulances should $unbury$ other agents before civilians; this ensures the most agents are enabled. Ambulances should $unbury$ civilians before attempting to $rescue$ them. Ambulances should $rescue$ civilians before $exploring$ new buildings. These goal orderings are specified in $\prec$ at the root node of the GLN, which means they propagate through all subgoals.

At the start of the demonstration, the CognitiveCycle updates WorkingMemory (cf. Figure 3) that there are new buildings and roads to explore. A FORMULATE strategy for each goal type creates goals to explore these entities. These goals require an agent to process them, so they remain in FORMULATED until an agent is assigned. Once assigned, the goal moves to DISPATCHED. When a building is explored by a vehicle entering it, it is FINISHED and then DROP removes them from memory.

6 Related Work

The inclusion of a temporal interval and ordering constraints in the GLN is reminiscent of a Simple Temporal Network (STN) (Dechter, Meiri, and Pearl 1991) and its extensions for dispatchable execution (e.g., (Muscettola et al. 1998; Tsamardinos, Muscettola, and Morris 1998; Kim, Williams, and Abramson 2001). It should be straightforward to extend the GN with an STN and incorporate standard consistency checking algorithms.

The EXECUTIONSTATEMENT is similar to the program step described in (Ghallab, Nau, and Traverso 2016, 98ff). where a program step can indicate execution state using RUNNING, DONE, or FAILED. There are many examples of more sophisticated execution state machines (e.g., (Niemueller, Hofmann, and Lakemeyer 2019; Estlin et al. 2006)) that we will integrate in future work.

Lima et al. (2020), during execution, also converts a totally-ordered plan into partially-ordered plan by relaxing causal

[7] `https://nms.kcl.ac.uk/planning/software/popf.html`

links in ROSPlan (Cashmore et al. 2015). The partially-ordered plan is then translated into a *set* of totally-ordered plans that can be flexibility selected between to take advantage of non-determinism in the environment. Our plan relaxation preserves causal sequencing and focuses on parallel execution of the plan.

A variety of Cognitive Architectures have been studied for decades. For example, in addition to SOAR, ACT-R, Sigma (Laird, Lebiere, and Rosenbloom 2017), there is the MIDCA system (Cox, Dannenhauer, and Kondrakunta 2017) and Icarus (Choi and Langley 2018). Our work complements these by integrating a simple Cognitive Cycle with the GLN.

7 Summary

We formalized the Goal Lifecycle Network (GLN), which synthesizes the literature on Hierarchical Goal Networks and goal lifecycles while updating both to use a state-variable representation. We then demonstrated GLNs being used in three robotics applications. Future work will extend this analysis to other theories on goal management (e.g., (Thangarajah et al. 2010; Harland et al. 2014; Cox, Dannenhauer, and Kondrakunta 2017)) and develop a richer execution model such as those by Niemueller et al. (2019) or Estlin et al. (2006).

Acknowledgments

The authors thank NRL and ONR for funding this work.

References

Aha, D. W. 2018. Goal Reasoning: Foundations, Emerging Applications, and Prospects. *AIMag* 39(2):3–24.

Akin, H. L.; Ito, N.; Jacoff, A.; Kleiner, A.; Pellenz, J.; and Visser, A. 2012. Robocup rescue robot and simulation leagues. *AI magazine* 34(1):78.

Alford, R.; Shivashankar, V.; Roberts, M.; Frank, J.; and Aha, D. W. 2016. Hierarchical planning: Relating task and goal decomposition with task sharing. In *Proc. IJCAI*, 3022–3028. New York, New York, USA: AAAI Press.

Cashmore, M.; Fox, M.; Long, D.; Magazzeni, D.; Ridder, B.; Carrera, A.; Palomeras, N.; Hurtos, N.; and Carreras, M. 2015. ROSPlan: Planning in the robot operating system. In *Proc. ICAPS*.

Choi, D., and Langley, P. 2018. Evolution of the Icarus Cognitive Architecture. *Cog. Sys. Research* 48:25–38.

Coles, A.; Coles, A.; Clark, A.; and Gilmore, S. 2011. Cost-sensitive concurrent planning under duration uncertainty for service-level agreements. In *Proc. ICAPS*.

Cox, M.; Dannenhauer, D.; and Kondrakunta, S. 2017. Goal operations for cognitive systems. *Proc. AAAI* 31(1).

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

Dvorak, F.; Bit-Monnot, A.; Ingrand, F.; and Ghallab, M. 2014. A flexible ANML actor and planner in robotics. In *Proc. ICAPS PlanRob Workshop, Portsmouth, United States*.

Estlin, T.; Jonsson, A.; Pasareanu, C.; Simmons, R.; Tso, K.; and Verma, V. 2006. Plan Execution Interchange Language (PLEXIL). Tech. Rep. NASA/TM-2006-213483, NASA.

Eyerich, P.; Mattmüller, R.; and Röger, G. 2009. Using the context-enhanced additive heuristic for temporal and numeric planning. In *Proc. ICAPS*.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *J. of Art. Intell. Res.* 20:61–124.

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

Harland, J.; Morley, D. N.; Thangarajah, J.; and Yorke-Smith, N. 2014. An operational semantics for the goal life-cycle in bdi agents. *Proc. AAMAS* 28:682–719.

Hawes, N. 2011. A survey of motivation frameworks for intelligent systems. *Art. Intel. J.* 175(5-6):1020–1036.

Kim, P.; Williams, B. C.; and Abramson, M. 2001. Executing reactive, model-based programs through graph-based temporal planning. In *Proc. IJCAI*, 487–493.

Laird, J. E.; Lebiere, C.; and Rosenbloom, P. S. 2017. A standard model of the mind. *AI Magazine* 38(4):13–26.

Lima, O.; Cashmore, M.; Magazzeni, D.; Micheli, A.; and Ventura, R. 2020. Robust execution of deterministic plans in non-deterministic env'ts. In *ICAPS IntEx-GR Workshop*.

Muscettola, N.; Nayak, P.; Pell, B.; and Williams, B. C. 1998. Remote agent: To boldly go where no AI system has gone before. *Art. Intell.* 103(1-2):5–47.

Niemueller, T.; Hofmann, T.; and Lakemeyer, G. 2019. Goal reasoning in the CLIPS executive for integrated planning and execution. In *Proc. ICAPS*, 754–763.

Preiss, J. A.; Honig, W.; Sukhatme, G. S.; and Ayanian, N. 2017. Crazyswarm: A large nano-quadcopter swarm. In *Proc. ICRA*, 3299–3304.

Roberts, M.; Shivashankar, V.; Alford, R.; Leece, M.; Gupta, S.; and Aha, D. 2016. Goal reasoning, planning, and acting with actorsim, the actor simulator. In *Poster Proc. ACS*.

Sheh, R.; Schwertfeger, S.; and Visser, A. 2016. 16 years of robocup rescue. *KIJ* 30(3):267–277.

Shivashankar, V.; Kuter, U.; Nau, D.; and Alford, R. 2012. A hierarchical goal-based formalism and algorithm for single-agent planning. In *Proc. of AAMAS*, volume 2, 981–988.

Shivashankar, V.; Alford, R.; Kuter, U.; and Nau, D. 2013. The GoDeL planning system. In *Proc. IJCAI*, 2380–2386.

Sucan, I. A.; Moll, M.; and Kavraki, L. E. 2012. The open motion planning library. *IEEE Robotics and Automation Magazine* 19(4):72–82.

Thangarajah, J.; Harland, J.; Morley, D. N.; and Yorke-Smith, N. 2010. Operational behaviour for executing, suspending, and aborting goals in bdi agent systems. In *DALT*.

Tsamardinos, I.; Muscettola, N.; and Morris, P. 1998. Fast transformation of temporal plans for efficient execution. In *Proc. AAAI*, 254–261. USA: AAAI.

Veloso, M. M.; Pérez, M. A.; and Carbonell, J. G. 1990. Nonlinear planning with parallel resource allocation. In *Proc. DARPA Workshop of Innovative Approaches to Planning, Scheduling and Control*. Morgan Kaufmann.