ACTORSIM: A toolkit for studying Goal Reasoning, Planning, and Acting

Mark Roberts[1] **Ron Alford**[2] **Vikas Shivashankar**[3] **Michael Leece**[4] **Shubham Gupta**[5] **David W. Aha**[6]

[1]NRC Postdoctoral Fellow; Naval Research Laboratory (Code 5514); Washington, DC | mark.roberts.ctr@nrl.navy.mil
[2]MITRE; McLean, VA | ralford@mitre.org
[3]Knexus Research Corp.; Springfield, VA | vikas.shivashakar@knexusresearch.com
[4]Dept. of Computer Science; Univ. of California Santa Cruz; Santa Cruz, CA | mleece@soe.ucsc.edu
[5]Thomas Jefferson High School for Science and Technology; Alexandria, VA
[6]Naval Research Laboratory (Code 5514); Washington, DC | david.aha@nrl.navy.mil

Abstract

Goal reasoning is maturing as a field, but lacks a unified model and common implementation that researchers can build from. This paper presents three contributions. First, it formalizes goal reasoning with crisp semantics by extending a recent formalism that blends goal-network and task-network planning. Second, it describes an open source package, called the Actor Simulator (ACTORSIM), that has been used in activity planning for robotics and partially implements the semantics of the formal model. Third, it presents a new study applying ACTORSIM and goal reasoning to the game of Minecraft. The study examines the role of learning from experience to improve goal selection and reveals that simple mechanisms for capturing experience are adequate for the problem we study.

1 Introduction[1]

Goals are a unifying structure for designing and studying intelligent systems, including robotic systems, which may perform goal reasoning to manage long-term behavior, anticipate the future, select among priorities, commit to action, generate expectations, assess tradeoffs, resolve the impact of notable events, or learn from experience. If a goal is an objective a robot wishes to achieve or maintain, then planning is deliberating on what action(s) best accomplish the objective, acting is deliberating on how to perform each action of a plan, and goal reasoning is deciding what goal(s) to pursue given trade-offs in dynamic, possibly adversarial, environments. Thus, goal reasoning is a critical component for enabling more responsive and capable autonomy.

Researchers have examined a variety of goal reasoning topics (Vattam et al., 2013), including studies on Goal-Driven Autonomy (Klenk et al., 2013; Munoz-Avila et al., 2010; Dannenhauer et al., 2015), goal formulation (Wilson, Molineaux, and Aha 2013), goal motivators (Munoz-Avila, Wilson, and Aha 2015), goal recognition (Vattam and Aha 2015), goal prioritization (Young and Hawes 2012), explanation generation (Molineaux and Aha 2014), and agent-oriented programming (Thangarajah et al., 2010; Harland et al., 2014; De Giacomo et al., 2016). Some studies have proposed models for specific aspects of goal reasoning, namely planning and acting (Thangarajah et al., 2010, Harland et al., 2014, Cox et al. 2016), while one study by Roberts et al. (2015b) adds goal formulation and goal selection to complete the entire lifecycle but lacks semantics . Four workshops[2] provide a more complete survey of the area. As this area of research matures, it can be enriched by more comprehensive studies using publicly available systems that implement a clear semantics.

We describe the Actor Simulator, ACTORSIM, which is a general platform for conducting studies of goal reasoning in simulated environments. Although goal reasoning has strong ties to planning, acting, and robotics, it is an understudied area of research partly because there exists no publicly available language, definition, and generic implementation. We draw inspiration from the literature in planning, where 15 years of international competitions has blossomed into a research ecosystem of nearly 100 open source planning systems and hundreds of planning benchmarks in a standardized language. Despite this progress, many planning systems focus on a single actor operating within a static environment and with static objectives. ACTORSIM can facilitate studies in which these constraints are relaxed.

ACTORSIM implements the goal reasoning model of Roberts et al. (Roberts et al. 2015a). We view goal reasoning as leveraging the work of Ghallab, Nau, and Traverso (Ghallab, Nau, and Traverso 2014; Nau, Ghallab, and Traverso 2015), wherein deliberation takes place on (1) descriptive models of *what* to accomplish (e.g., a goal to be in room_a might be decomposed into the subgoals to be near a door for room_a, achieve $\text{open}(\text{door}_{to-A})$ if applicable, and then perform the task $\text{enter}(\text{room}_a)$), and (2) operational models of *how* to perform a task (e.g., opening the door is a sequence of subtasks such as: determine the type of door handle, grasp the handle, and push (or pull) the door). Thus, the deliberation in such systems resembles a task network with goals interspersed. Similarly, we argue that goal reasoning is a kind of hybrid task-goal planning that supports the larger cycle of Planning and Acting by allowing an actor to determine and prioritize its goals dynamically.

Contributions. Our objective in this paper is to foster studies of goal reasoning by presenting:

[1]A more recent version of this paper will appear in the Proceedings of the 4th Annual Conference on Advances in Cognitive Systems; see Roberts et al. (to appear) for details.

[2]The latest workshop is described at http://makro.ink/ijcai2016grw/

A **formal model** of goal reasoning and its semantics. This model extends previous work by Roberts et al. (2015) and builds on a hybrid model of planning called Goal-Task Network (GTN) planning (Anonymous, under review), which blends Hierarchical Task Network (HTN) planning with Hierarchical Goal Network (HGN) planning. This hybrid model allows us to seamlessly intermix task and goal networks with state-based planning concerns, which is critical in a system that performs goal reasoning and deliberation (Ghallab, Nau, and Traverso 2014).

An **open source platform**, called ACTORSIM[3], that implements this formal model. ACTORSIM's initial design spurred from work on robotic applications to Foreign Disaster Relief operations (Roberts et al. 2015b) and has since been extended to several other domains. We briefly summarize how ACTORSIM has supported these studies and our future plans for integration with more sophisticated simulators such as ROS or Gazebo.

The **application** of ACTORSIM to Minecraft, with preliminary results showing that (1) learning from structured experience to select subgoals improves behavior for a simple navigation task, (2) gathering evidence showing expert knowledge is useful but not essential for effective decision making in this task, and (3) costly random knowledge gathering, as typically performed in unsupervised learning, is best used only to broaden structured knowledge. Our results complement existing studies on how to gather and learn from experience and demonstrate that goal networks can overcome some limitations of action selection approaches.

The paper proceeds in two parts. The first part establishes a model of goal reasoning. We briefly describe our notation (Section 2) and a formalism called *Goal-Task Network* (GTN) planning (Section 2.1) that we will apply to define the semantics of goal reasoning. Section 3 formalizes the model and semantics that ACTORSIM implements.

The second part provides a snapshot of ACTORSIM (Section 4) and then describes an implementation of ACTORSIM for the game of Minecraft (Section 5). We present a pilot study that highlights the benefits of applying learning to goal selection in a simple maze problem (Section 6). Finally, we describe other ACTORSIM connectors we have developed (Section 7), related work (Section 8), and conclude with future work objectives.

2 Preliminaries

Ghallab et al. (2014) and Nau et al. (2015) point out that planning and acting systems must often deliberate about both descriptive and operational models. Descriptive models detail *what* actions would accomplish a goal (e.g., "plans"), while operational models detail *how* to accomplish it; (e.g., "tasks" or "procedures"). Thus, a hybrid model that combines state-based planning and hierarchical planning is needed.

Let $\mathcal{L}$ be a propositional language. We partition $\mathcal{L}$ into *external state* $s \subset \mathcal{L}_{external}$ relating to an agent's belief about the world, where the set of all external states is $S = 2^{\mathcal{L}_{external}}$, and *internal state* $z \subset \mathcal{L}_{internal}$ relating to internal decisions and processes of the agent, where the set of all internal states is $Z = 2^{\mathcal{L}_{internal}}$. $\mathcal{L} = \mathcal{L}_{external} \cup \mathcal{L}_{internal}$, where $\mathcal{L}_{external} \cap \mathcal{L}_{internal} = \emptyset$.

Let $\mathcal{T}$ be a set of task names represented as propositional symbols not appearing in $\mathcal{L}$ (i.e., $\mathcal{L} \cap \mathcal{T} = \emptyset$), and let O and C be a partition of $\mathcal{T}$ ($O \cup C = \mathcal{T}$, $O \cap C = \emptyset$). O denotes the set of *primitive tasks* that can be executed directly, while C represents compound or *non-primitive* tasks that need to be recursively decomposed into primitive tasks before they can be executed.

We augment the model of online planning and execution by Nau (2007) with a goal reasoning loop (cf. Figure 1). The world is modeled as a state transition system $\Sigma = (S, A, E, \delta)$ where S is a set of states that represent facts in the world as above, $A = (a_1, a_2, ...)$ are the allowed actions of the Controller, $E = (e_1, e_2, ..)$ is a set of exogenous events, and $\delta : S \times (A \cup E) \rightarrow S$ is a state transition function. Let s_{init} denote the initial state and S_g denote the set of allowed goal states. The *classical* planning problem is stated: Given $\Sigma = (S, A, \delta)$, s_{init} and S_g, find a sequence of actions $\langle a_1, a_2, .., a_k \rangle$ such that $s_1 \in \delta(s_{init}, a_1), s_2 \in \delta(s_1, a_2), .., s_k \in \delta(s_{k-1}, a_k)$ and $s_k \in S_g$. Thus, the actor seeks a set of transitions from s_{init} to one of a set of goal states $S_g \subset S$.

We call the goal reasoner in Figure 1 the GRPROCESS and assume the Controller only executes one action x_j at a time, returning PROGRESS$_j$ to update progress, SUCCESS$_j$ for completion, and FAIL$_j$ for failure. A goal memory stores goals that transition through the goal lifecycle, which we will define more fully in §3.2. We simplify the discussion by considering only achievement goals even though the model and ACTORSIM can represent maintenance goals.

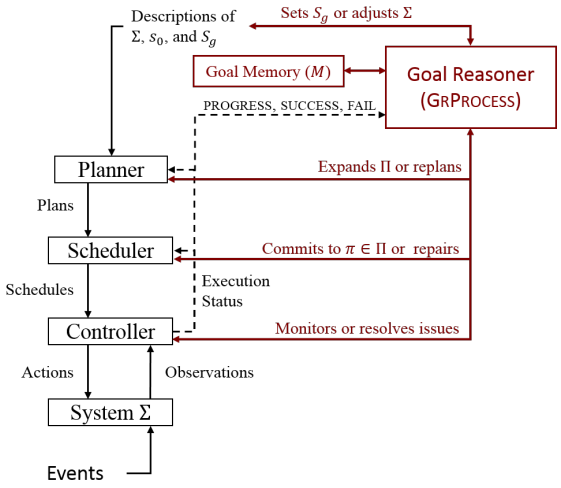

Figure 1: Relating goal reasoning with online planning, where the GRPROCESS works with a goal memory to monitor and modify the goals or planning model of the system. The goal memory stores goal nodes that transition according to the goal lifecycle described later.

[3]Available at `http://makro.ink/actorsim`

2.1 Goal-Task Network (GTN) Planning

Alford et al (2016) model both hierarchical task and goal planning in a single framework called Goal-Task Network (GTN) planning, which was partly inspired by conversations with Ghallab, Nau, & Traverso following their position paper on Planning and Acting (Ghallab, Nau, and Traverso 2014). GTN planning augments the notation of (Geier and Bercher 2011) with goal decomposition from HGN planning (Shivashankar et al., 2012) and SHOP2-style method preconditions (Nau et al., 2003). While HTN planning is over partially-ordered multisets of *task names* from $\mathcal{T}$ and HGN planning is over totally-ordered subgoals in $\mathcal{L}$, GTN elegantly models both. The rest of this section summarizes Alford et al. (2016) as it relates to the goal reasoning model we introduce.

A *goal-task network* is a tuple $(I, \prec, \alpha)$ where I is a set of instance symbols that are placeholders for task names and goals, $\prec \subset I \times I$ is a partial order on I, and $\alpha : I \rightarrow \mathcal{L} \cup \mathcal{T}$ maps each instance symbol to a goal or task name. An instance symbol i is *unconstrained* if no symbols are constrained to be before it ($\forall_{i' \in I}\ i' \not\prec i$) and *last* if no symbols are constrained to be after it ($\forall_{i' \in I}\ i' \prec i$). A symbol i is a *task* if $\alpha(i) \in \mathcal{T}$ and is a *goal* if $\alpha(i) \in \mathcal{L}$; recall that $\mathcal{L}$ and $\mathcal{T}$ are disjoint.

Methods We distinguish the methods of a GTN by the kind of symbol they decompose. A *task method* m_t is a tuple (n, χ, gtn) where $n \in C$ is a non-primitive task name, $\chi \in \mathcal{L}$ is the precondition of m_t, and gtn is a goal-task network over $\mathcal{L}$ and $\mathcal{T}$. m_t is *relevant* to a task i in $(I, \prec, \alpha)$ if $n = \alpha(i)$. m_t is a specific decomposition of a task n into a partially-ordered set of subtasks and subgoals, and there may be many such methods. A *goal method* m_g, similarly, is a tuple (g, χ, gtn) where $g, \chi \in \mathcal{L}$ are the goal and precondition of m_g and gtn is a goal-task network. m_g is *relevant* to a subgoal i in $(I, \prec, \alpha)$ if at least one literal in the negation-normal form (NNF) of g matches a literal in the NNF of $\alpha(i)$ (i.e., accomplishing g ensures that part of $\alpha(i)$ is true). By convention, $gtn = (I, \prec, \alpha)$ has a last instance symbol $i \in I$ with $\alpha(i) = g$ to ensure that m_g accomplishes its own goal.

Operators An *operator* o is a tuple (n, χ, e) where $n \in O$ is a primitive task name (assumed unique to o), χ is a propositional formula in $\mathcal{L}$ called o's precondition (or $prec(o)$), and e is a set of literals from $\mathcal{L}$ called o's effects. We refer to the set of positive literals in e as $add(o)$ and the negated literals as $del(o)$. An operator is *relevant* to primitive task i_t if $n = \alpha(i_t)$ and to a subgoal i_g if the effects of o contain a matching literal from the NNF of $\alpha(i_g)$. A set of operators $\mathcal{O}$ forms a transition (partial) function $\gamma : 2^{\mathcal{L}} \times \mathcal{O} \rightarrow 2^{\mathcal{L}}$ as follows: $\gamma(s, o)$ is defined iff $s \models prec(o)$ (the precondition of o holds in s), and $\gamma(s, o) = (s \setminus del(o)) \cup add(o)$.

GTN Nodes and Progression Operations Let $N = (s, gtn)$ be a *gtn-node* where s is a state and gtn is a goal-task network. A *progression* transitions a node N by applying one of four progression operations: operator application (A), task decomposition (D_t) goal decomposition (D_g), or release (G). Let $P = \{A, D_t, D_g, G\}$ represent any of these four operations (when the context is clear we write D for either D_t or D_g). Then $N \rightarrow_P N'$ denotes a single progression operation from N to N‘, while $N \rightarrow_P^* N''$ denotes a progression sequence from N to N''. Here we only summarize these operations, although their semantics are defined by Alford et al. (2016).

Operator application, $(s, gtn) \xrightarrow{i,o}_A (s', gtn')$, applies an operator o to a node (s, gtn), with $gtn = (I, \prec, \alpha)$ and is defined if $s \models prec(o)$ and o is relevant to an unconstrained instance symbol i in gtn. If i is a primitive task with task name n, then this corresponds to primitive task application in HTNs. If i is instead a relevant goal task, this corresponds to primitive task application in HGNs; in this case, $gtn' = gtn$, and the subgoal remains while the state changes.

Goal decomposition, $(s, gtn) \xrightarrow{i,m}_D (s, gtn')$, for an unconstrained subgoal i by a relevant goal method $m = (g_m, \chi, gtn_m)$ is defined whenever $s \models \chi$. It *prepends* i with gtn_m.

Task decomposition, $(s, gtn) \xrightarrow{i,m}_D (s, gtn')$, for an unconstrained task i by a relevant task method $m = (c, \chi, gtn_m)$ is defined whenever $s \models \chi$. It expands i in gtn, replacing i with the network gtn_m.

Goal release, $(s, gtn) \xrightarrow{i}_G (s, gtn')$, for an unconstrained subgoal i is defined whenever $s \models \alpha(i_g)$. It can remove a subgoal whenever it is satisfied by s.

GTN Planning Problems and Solutions A *gtn-problem* is a tuple $P = (\mathcal{L}, \mathcal{O}, \mathcal{M}, N_0)$, where $\mathcal{L}$ is propositional language defining the operators ($\mathcal{O}$) and methods ($\mathcal{M}$), N_0 is the initial node consisting of the initial state s_0, and gtn_0 is the initial goal-task network. O and C are implicitly defined by $\mathcal{O}$ and $\mathcal{M}$. A problem P is *solvable* under GTN semantics iff there is a progression $N_0 \rightarrow_P^* N_k$, where $N_k = (s_k, gtn_\emptyset)$, s_k is any state, and $gtn_\emptyset$ is the empty network.

Solutions are distinguished by two kinds of plans that depend on whether the world state is changed via operator application. The subsequence of *operator applications* of a progression sequence is a *plan* for P, since such operations modify world state. A *gtn-plan* for P is consists of all progression operators, since this sequence captures the entire set of progressions that must occur for a valid solution. The GRPROCESS produces gtn-plans as explained in the following section.

3 A Goal Reasoning Model

To arrive at a goal reasoning model, we blend GTN semantics with the goal lifecycle in Figure 2 to define a semantics for the GRPROCESS we have partially implemented in ACTORSIM. We begin by extending the online planning model of §2 to model the GR actor as a state transition system $\Sigma_{gr} = (M, R, \delta_{GR})$, where M is the goal memory, R is a set of refinement strategies, and $\delta_{gr} : M \times R \rightarrow M'$ is the goal-reasoning transition function. We next define these components.

3.1 Nodes, Progression, and the Goal Memory

The goal memory stores goal nodes. A *goal node* is a tuple $\mathcal{N} = (g_i, N, C, o, X, x, q)$ where: $g_i \subset \mathcal{L}$ is the goal to be achieved; $N = (s, gtn)$ is a gtn-node for g_i; Con is the set of constraints on g_i and gtn; o is the current mode of g_i, defined below; X is the set of expansions that could achieve g_i, defined below; $x \in X$ is the committed expansion along with any applicable execution status; and q is a vector of quality metrics. Metrics could be domain-dependent (e.g., priority, cost, value, risk, reward) and are associated with achieving g_i. An important domain-independent metric, *inertia*, stores the number of refinements applied to $\mathcal{N}$. Dotted notation indicates access to $\mathcal{N}$'s components, e.g., $\mathcal{N}.N := (s, gtn')$ indicates that the gtn-node gtn of $\mathcal{N}$ has been updated to gtn'.

Similar to GTN planning, progressions modify components of $\mathcal{N}$; we call these the refinement strategies R. Let χ be a set of preconditions and $r \in R$ denote a progression operator for $\mathcal{N}$. Then a refinement $r = (\mathcal{N}, \chi)$ transitions one or more components of $\mathcal{N}$ to $\mathcal{N}'$ and is written $\mathcal{N} \xrightarrow{N,C,o,X,q}_R \mathcal{N}'$. Refinement sequences from $\mathcal{N}$ to $\mathcal{N}''$ are written $\mathcal{N} \rightarrow^*_R \mathcal{N}''$. Preconditions χ come from either the goal lifecycle discussed below or domain-specific requirements for a specific world state or specific events before a refinement can transition.

The *goal memory* $M = \{\mathcal{N}_1, \mathcal{N}_2, .., \mathcal{N}_m\}$ for $m \geq 0$ holds the active goal nodes for the GRPROCESS. Most refinements modify the goal memory by modifying a node within memory, in which case we write $M \rightarrow_R M'$ for a single strategy application resulting in M' and $M \rightarrow^*_R M''$ for a sequence of applied strategies resulting in M''.

3.2 Operations and Semantics: Refinement Strategies (R)

Figure 2 displays the possible refinement strategies, where an actor's decisions consist of applying one or more refinements from R (the arcs) to transition $\mathcal{N}$ between modes (rounded boxes). Strategies are denoted using small caps (e.g., FORMULATE) with the modes in monospace (e.g., `FORMULATED`). For the remainder of this section, we detail semantics for many of these strategies. We shorten the discussion by omitting quality metrics $\mathcal{N}.q$ but leave the q above the progression to indicate that at least inertia is modified. For example, every refinement $\mathcal{N} \xrightarrow{q}_R \mathcal{N}'$ results in $\mathcal{N}'.q.inertia \mathrel{+}= 1$ indicating increased refinement effort on $\mathcal{N}$. GRPROCESS may favor nodes with higher inertia by pushing them toward completion or limiting further processing on them.

For the remainder of this section, we detail the forward sequence of a single goal node through the lifecycle of Figure 2, which consists of goal formulation (via FORMULATE), goal selection (SELECT), planning (EXPAND and COMMIT), plan execution (DISPATCH, MONITOR, EVALUATE), and resolving execution events (FINISH and the suite of RESOLVE-BY strategies, which are displayed as annotations to dashed lines). Finally, we explain two strategies that can be applied at any time: DROP and PROCESS (not shown in this figure).

Goal formulation and Goal Selection Two important decisions for GRPROCESS concern determining which goals to create (i.e., FORMULATE) and which to pursue (i.e., SELECT).

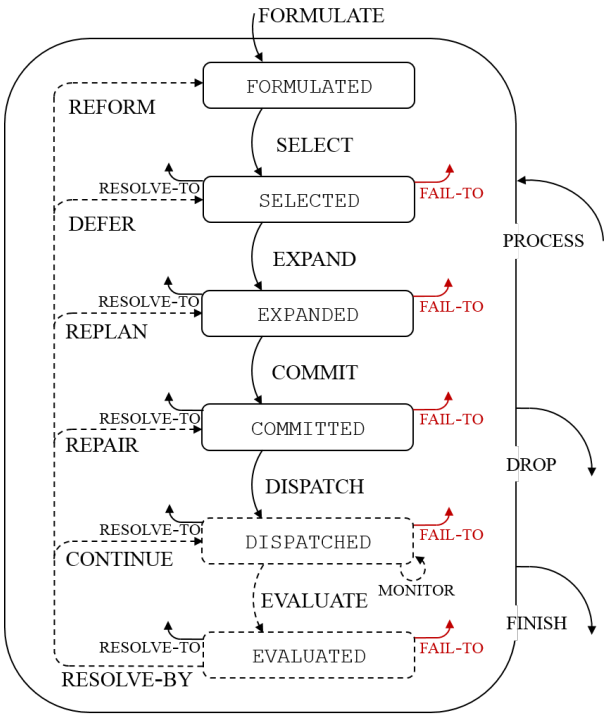

Figure 2: The goal lifecycle. Refinement strategies (arcs) denote possible decision points of an actor, while modes (rounded boxes) denote the status of a goal (set) in the goal memory.

FORMULATE adds a new goal to the goal memory, written $M \xrightarrow{g,\mathcal{N}}_{\text{FORM}} M'$ for a new goal g, its corresponding node $\mathcal{N}$, the goal memory M before the application, and M' the revised memory. The result of applying FORMULATE is: $\mathcal{N}.g = g$; $\mathcal{N}.N = (s_{current}, gtn_g)$; $\mathcal{N}.Con = \emptyset$; $\mathcal{N}.o =$ `FORMULATED`; $\mathcal{N}.X = \emptyset$; $\mathcal{N}.x = nil$; $\mathcal{N}.q.inertia = 1$; and $M' = M \cup \mathcal{N}$.

SELECT transitions $\mathcal{N}.o$ from `FORMULATED` to `SELECTED`, written $\mathcal{N} \xrightarrow{o,q}_{\text{SEL}} \mathcal{N}'$. It allows GRPROCESS to determine which goal nodes move forward and which remain `FORMULATED`. In a GRPROCESS where $|M| \leq k$ is bound to no more than k goals, SELECT can limit extensive processing on nodes. Many nodes trivially transition: $\mathcal{N}'o :=$ `SELECTED`.

Planning Classical planning systems often make strong assumptions about the kind of plan required (i.e., the optimal plan), the number (i.e., usually one), and the nature of execution (i.e., actions are deterministic and atomic). In contrast, a GRPROCESS may explore alternative plans and commit to one after further deliberation. We define an *expansion* to mean any kind of plan to achieve a goal. While we focus on state transitions in Σ or Σ_{gr}, expansions more generally include motion planning, trajectory planning, reactive planning, etc., as often used in robotics applications.

EXPAND, written $\mathcal{N} \xrightarrow{o,X,q}_{\text{EXP}} \mathcal{N}'$, generates expansions (i.e., gtn-plans) via operator application, task decomposition, and goal decomposition from §2.1. Consider a progression $\pi = N_0 \rightarrow_P^* N_k$, where $N_k = (s_k, gtn_\emptyset)$, s_k is any state, and $gtn_\emptyset$ is the empty network. Recall from §2.1 that such a progression is a solution to a GTN problem and was called a *gtn-plan*. EXPAND generates k expansions such that $x^1, x^2, .., x^k \in X$, $|X| > 0$, and $x^1 = \pi^1, .., x^k = \pi^k$ are the available expansions. The result is: $\mathcal{N}'.o \coloneqq$ EXPANDED and $\mathcal{N}'.X \coloneqq \{x^1, .., x^k\}$.

COMMIT chooses one expansion from $\mathcal{N}.X$ for Controller execution and is written $\mathcal{N} \xrightarrow{o,q,x}_{\text{COM}} \mathcal{N}'$. The result is: $\mathcal{N}'.o \coloneqq$ COMMITTED and $\mathcal{N}'.x \coloneqq x^c$ for some $1 \leq c \leq k$.

Plan Execution The Controller executes the steps in $\mathcal{N}.x$ until no more steps remain or a step fails; $\mathcal{N}$ is DISPATCHED during this progression. Some expansions (e.g., goal or task decomposition) are internal to the goal memory and do not result in external actions of the actor. In the case of decomposition, a node remains DISPATCHED until its subgoals or subtasks are completed. Other expansions (e.g., operator application) result in external actions by the Controller during execution. Plan execution consists of DISPATCH, MONITOR, and EVALUATE.

DISPATCH, written $\mathcal{N} \xrightarrow{o,N,q}_{\text{DISP}} \mathcal{N}'$, applies the steps of the progression within $\mathcal{N}.x$. First, the goal node transitions: $\mathcal{N}'.o \coloneqq$ DISPATCHED. Then, the GRPROCESS steps through the expansion $\mathcal{N}.x$. Suppose that $\mathcal{N}.x$ points to the expansion $x = N_0 \rightarrow_P^* N_k$ and that an index $0 < j \leq k$ indicates the step of the progression such that $N_{j-1} \rightarrow_P^j N_j$. For k steps in x and each step x_j for $0 < j \leq k$, the result is: $\mathcal{N}.N_{j-1} \rightarrow_P^j \mathcal{N}'.N_j$. How the GRPROCESS applies x_j depends on specified operation (cf. §2.1): *Operator Application* applies operator o to the instance symbol i. This application results the Controller executing i. *Task Decomposition* applies method m to a compound task i, written $\xrightarrow{i,m}_D$, such that $\mathcal{N}.N.gtn$ is progressed. *Goal Decomposition* applies method m to a goal i, written $\xrightarrow{i,m}_D$, such that $\mathcal{N}.N.gtn$ is progressed, resulting in new subgoals being added to the goal memory M. Let there be t new subgoals resulting from applying m to i, labeled $(g_{i1}, g_{i2}, .., g_{it})$. For goal g_{ij} where $0 < j \leq t$, then FORMULATE(g_{ij}) is called, resulting in $M \xrightarrow{g_{ij},\mathcal{N}}_{\text{FORM}} M'$.

MONITOR, if enabled, proactively checks on the status of $\mathcal{N}.x_j$. If the status is FAIL or is not meeting expectations, then EVALUATE is called. Nominal status only modifies the inertia.

EVALUATE, written $\mathcal{N} \xrightarrow{o,q}_{\text{EVAL}} \mathcal{N}'$, processes events that impact $\mathcal{N}$ during execution, which might include execution updates or unanticipated anomalies. This strategy allows a goal node to signal track that its execution is impacted: $\mathcal{N}'.o \coloneqq$ EVALUATED.

Resolving Notable Events A *notable event* is one that impacts $\mathcal{N}$. A number of possible strategies relate to such events and some are relevant from particular modes. Often the goal determines for itself whether an event is noteworthy, which simplifies the encoding of strategies for a domain. However, in more complex cases another deciding process may arbitrate this determination. Resolution strategies can roughly be divided into those that occur during execution (shown as dashed lines in Figure 1), those that are related to error conditions, and those that occur outside of executions or errors.

PROCESS may be called in any node. It is the means by which external processes or the GRPROCESS notify a goal about an event and allow the goal to determine whether the event is significant. In many cases, an event can be disregarded and the only the inertia is incremented. If the node is DISPATCHED then the event may impact the execution of a step x_j. The impact of the event may be positive (e.g., completion of x_j), neutral (e.g., x_j is progressing as expected) or negative (e.g., the imminent or detected failure of x_j). In this case, $\mathcal{N}$ transitions to EVALUATED and there are several possible resolutions from this mode, as shown by the dashed RESOLVE-BY strategies of Figure 1.

RESOLVE-BY can only be called from EVALUATED and consists of a suite of strategies, which we only briefly describe. These strategies are distinct because the Controller may need to be notified. CONTINUE allows $\mathcal{N}$ to proceed without significant change to its members. ADJUST corrects the state models Σ or Σ_{GR} that would modify future planning. REPAIR modifies the current expansion x to x'. REEXPAND creates new expansions $\{x'^1, .., x'^k\}$ for the GRPROCESS to consider. DEFER returns $\mathcal{N}$ in a SELECTED mode and REFORMULATE returns $\mathcal{N}$in a FORMULATED mode. FAIL-TO is a failure mode that allows the GRPROCESS to return a goal to any previous mode for further processing. This strategy applies when a transition is attempted but fails. For example, if a plan cannot be generated then EXPAND may trigger FAIL-TO(SELECTED).

RESOLVE-TO is used when a notable event impacts a node but the impact is not deemed a failure. For example, if a plan has already been generated but the goal for a node is preempted, then the GRPROCESS may call RESOLVE-TO(FORMULATED) to unselect the goal. In contrast to RESOLVE-BY, these methods simply "park" $\mathcal{N}$ in the appropriate mode and will not otherwise modify the goal node. Such progressions may be useful for quickly pausing a goal.

DROP removes $\mathcal{N}$ from M such that $M' = M \setminus \{\mathcal{N}\}$. It is analogous to goal release (cf. §2.1).

FINISH, written $\mathcal{N} \xrightarrow{o,q}_{\text{FIN}} \mathcal{N}'$, indicates that execution is complete for this expansion. FINISH cannot complete if subgoals in gtn exist in the memory M. If x involved decomposition, then all subgoals or subtasks have been DROPed. If x involved operator application, then the Controller returned SUCCESS. This strategy does not remove $\mathcal{N}$ from M, which is performed by DROP.

3.3 Goal Reasoning Problems and Solutions

Let $P_{gr} = (\mathcal{L}, \mathcal{O}, \mathcal{M}, R_d, R_p, M_0)$ be a goal-reasoning problem where $\mathcal{L} = \mathcal{L}_{external} \cup \mathcal{L}_{internal}$ is the propositional language, $\mathcal{M}$ and $\mathcal{O}$ are defined as in Section 2.1, R_d is the default set of refinement strategies detailed in Section 3.2,

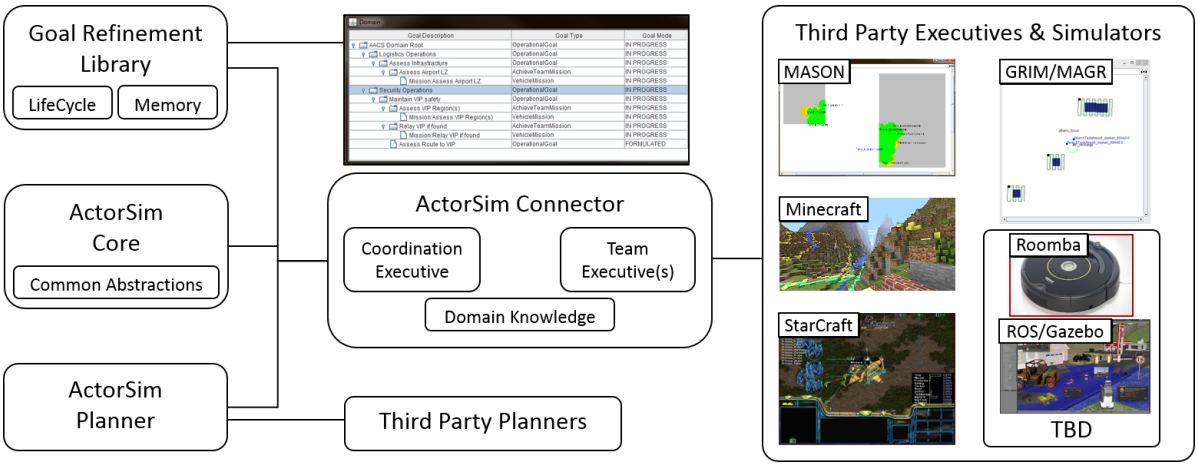

Figure 3: The Component Architecture of ACTORSIM

R_p is a set of refinement strategies provided by the domain designer, and M_0 is the initial goal memory.

We say that P_{gr} is solvable iff there is a progression $M_0 \rightarrow_R^* M_k$, where $M_k = \emptyset$.

4 The Actor Simulator

The Actor Simulator, ACTORSIM (Figure 3), is a partial implementation of the goal lifecycle of Roberts et al. (2015), which is described in Section 3.2. ACTORSIM complements existing open source planning systems with a standardized implementation of goal reasoning. It also provides links to simulators that can simulate multiple agents interacting within a dynamic environment.

ACTORSIM **Core** provides the interfaces and minimal implementations of the platform. It contains the essential abstractions that apply across any simulator. This component contains information about Areas, Locations, Actors, Vehicles, Symbols, Maps, Sensors, and configuration details.

ACTORSIM **Planner** contains the interfaces and minimal implementations for linking to existing open source planning systems. This component unifies Mission Planning, Task Planning, Path Planning, and Motion Planning. It currently includes simple, hand-coded implementations of these planners, although we envision linking this component to many open source planning systems.

ACTORSIM **Connector** links to existing simulators directly or through a network protocol. Currently supported simulators include George Mason University's MASON[4] and two computer game simulators: StarCraft and Minecraft. We envision links to common robotics simulators (e.g., Gazebo, ROS, OpenAMASE), additional game engines (e.g., Mario Bros., Atari arcade, Angry Birds), and existing competition simulators (e.g., RDDLSim). We may eventually link ACTORSIM to physical hardware.

ACTORSIM **Coordinator** (not shown in the figure) houses the interfaces that unify all the other components. This component contains abstractions for Tasks, Events, Human interface Interaction, Executives (i.e., Controllers), and Event Notifications. It uses Google's protocol buffers[5] for messaging between distributed components.

The **Goal Refinement Library** is a standalone library that is integral to ACTORSIM , but could be used on its own. It provides goal management and the data structures for transitioning goals throughout the system. This library contains the default implementations for goals, goal types, goal refinement strategies, the goal memory, domain loading, and domain design.

5 Overcoming Obstacles in Minecraft

We study goal reasoning in Minecraft, a popular game where a human player moves a character, named Steve, to explore a 3D virtual world while gathering resources and surviving dangers. Managing the complete game is challenging. The character holds a limited inventory to be used for survival. Resource blocks such as sand, dirt, wood, and stone can be crafted into new items, which in turn can be used to construct tools (e.g., a pickaxe for mining or shovel for digging) or structures (e.g., a shelter, house, or castle). Some blocks are dangerous to the character (e.g., lava or water). Hostile non-playing characters like a creeper or skeleton, generally called mobs, can damage the characters health. Steve can only fall two blocks without taking damage. We focus on the problem of navigating a much simpler subset of the world. The set of possible choices available to achieve even this *simple* goal is staggering; for navigating a 15x15 maze in Minecraft, Abel et al. (2014) estimate the state space to be nearly one million states.

Researchers have recently begun using the Minecraft game for the study of intelligent agents (Aluru et al. 2015). In previous work, researchers developed a learning architecture called the Brown-UMBC Reinforcement Learning and Planning (BURLAP) library, which they implemented in their variant of Minecraft, BURLAPCraft (Abel et al. 2015) BURLAPCraft allows a virtual player to disregard certain

[4] http://cs.gmu.edu/~eclab/projects/mason/

[5] https://developers.google.com/protocol-buffers/

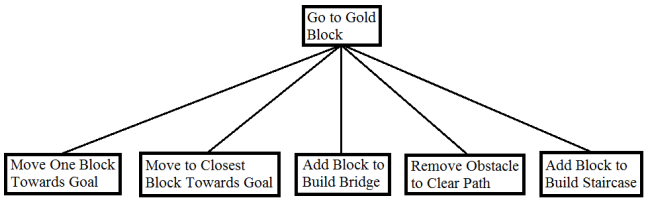

Figure 4: The gtn for the GRPROCESSin our study.

actions that are not necessary for achieving goals such as navigating a maze.

Similar to that research, we task the GRPROCESS , acting as a virtual player, with controlling Steve to achieve the goal of navigating to a gold block through an obstacle course. However, our technical approach differs from prior research. Our aim is to develop a GRPROCESS that can incorporate increasingly sophisticated goal-task networks and learned experience about when to apply them. At a minimum, this requires thinking about how to compose action primitives into tasks that the GRPROCESS can apply and linking these tasks into a gtn. Thus, we construct these tasks and build a gtn that uses them.

Figure 4 shows the gtn consisting of a top goal of moving to the gold block and the five descriptive subgoals that help the character lead to that objective. These subgoals do not contain operational knowledge. For example, preconditions on actions ensure that Steve will not violate safety by falling too far or walking into a pool of lava or water. For moving toward the goal, the block at eye level must be air, the block stepped on cannot be lava or water, and Steve cannot fall more than a height of two blocks. A staircase requires a wall with a height of two blocks and the ability to move backwards in order to place a block. Mining is only applicable if the obstacle has a height of three blocks.

The order of subgoal choice impacts performance. For example, suppose the subgoal to step forward is selected when lava is directly in front of Steve. Steve's Controller disallows this step because it violates safety and the subgoal will fail, which will require additional goal reasoning to resolve the failure.

Two features of our goal representation complement prior research in action selection (e.g., reinforcement learning or automated planning). First, we model the subgoal choice at descriptive level, assuming that committing to a subgoal results in an effective operational sequence (i.e., a plan) to achieve the goal. We rely on feedback of the Controller running the plan to resolve the subgoal. Second, the entire state space from start to finish is inaccessible to the GRPROCESSso it cannot simply perform offline planning or interleave full planning with online execution. Each obstacle course is distinct and there must be an interleaving of perception, goal reasoning, acting. Third, the operational semantics of committing to a subgoal are left to the Controller. Thus, the GRPROCESS must learn to rank the subgoals based on the current state using prior experience. Although random exploration is possible, we will present evidence that that such an approach is untenable, corroborating the findings of

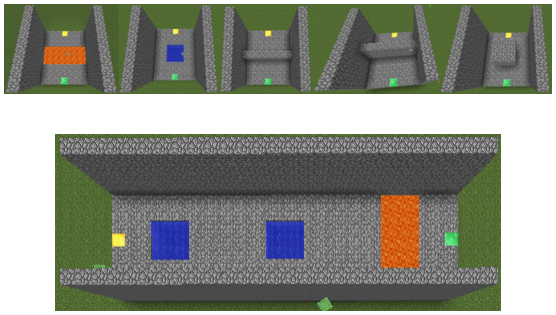

Figure 5: The section types (top) and a short obstacle course (bottom) where the GRPROCESS must traverse from the emerald block on the right to the gold block on the left while through a lava pit and two ponds.

Abel et al. (2015) that the state/action space is too large to explore without a bias.

The question, then, is how to bias the exploration in such a way as to speed up learning. *Our research hypothesis is that making effective choices at the* GTN *level can be done by learning from traces (i.e., examples) that lead to more efficient behavior, where improved efficiency is measured as getting to the goal in fewer steps or failing less frequently.* Our research focus in this paper is examining what kind of experience is most valuable. To this end, we demonstrate a pilot study that leverages three kinds of prior experience (completely random, ordered, and expert) to learning an effective subgoal selection policy.

We next describe how ACTORSIM connects to Minecraft and how we collect experience.

5.1 The Minecraft Connector in ACTORSIM

The Minecraft Connector integrates ACTORSIM abstractions with a reverse-engineered game plugin called the Minecraft Forge API (Forge), which provides methods for manipulating Minecraft. We implemented basic motion primitives such as looking, moving, jumping, and placing or destroying blocks. These motion primitives compose the operational plans for the five sub-goals: step forward, move closer, step up, mine, and bridge. Although some of this functionality was present in BURLAPCraft (Abel et al. 2015), our implementation better matches with the abstractions provided by ACTORSIM Core and ACTORSIM Coordinator.

We have simplified Steve's motions to be axis aligned. Steve always faces North and the maze is constructed such that the gold block is North of Steve in a straight line. Steve is 1.8 meters high; voxels in Minecraft are 1 meter square. So, Steve occupies roughly a 1x2 meter space. Steve interacts with a limited set of world objects: cobblestone, emerald, air, lava, water, and gold.

The Minecraft connector constructs the obstacle courses for our study. Figure 5 (top) shows the five sections the GRPROCESS may encounter: lava, pond, short wall (2 blocks high), tall wall (3 blocks high), and pillar (3 blocks high). Figure 5 (bottom) shows a maze composed of three sections. Steve begins at an emerald block on the right with

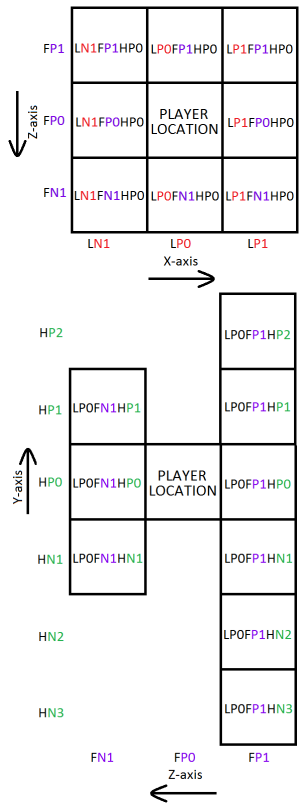

Figure 6: Observable blocks around Steve from the top view (top), where the player is facing "up" and the side view (bottom), where the player is facing to the right.

a goal of being at the gold block.

Each obstacle has an appropriate subgoal choice. For lava or pond, the best choice is a bridge; alternatively the GRPROCESS may also move closer and go around the pond. For the short wall, the best subgoal is to create a single stair and step up. For the tall wall or pillar, which are both three blocks high, the best subgoal is to mine through the wall; alternatively, the GRPROCESS may also move closer and go around the pillar.

Observations Figure 6 shows the set of states around Steve that the GRPROCESS can observe. These include the eight blocks directly around Steve's feet, the two blocks directly behind and in front of Steve, one block behind and below Steve, the block just above Steve's head to the front, and the block three down and in front of Steve. A state is labeled with a unique string using the relative position left/right (l), front/back (f), and height (h) with either a positive (p) or negative (n) offset, where zero is denoted as a positive number. Each state is assigned a unique string (shown in each box) to denote the world object in that position.

Collecting Traces of Experience We collect three kinds of traces for choosing these five subgoals that vary in how much state they consider. The **random** training procedure is worst-case baseline; it ignores state and selects a subgoal with uniform probability. The **ordered** training procedure selects the subgoals in the same order of Figure 4: step forward, move closer, bridge, mine, and step up; it also ignores state. If a subgoal is allowed by the Controller, the subgoal is applied. If not, it continues to the next subgoal in the ordering; on success selection restarts from the beginning of the order. The **expert** training procedure ensures most runs reach the gold as a best-case bound; it is hand-coded (by an author of this paper) and examines detailed state to select the best subgoal.

We collect traces from the random, ordered, and expert procedures, capturing the state, distance to the goal, sub-goal chosen, and whether the chosen subgoal succeeded. Both the random and ordered training procedures can fail to reach the gold. The expert procedure never fails to reach the gold but also represents extremely biased knowledge about which subgoal is appropriate.

6 Learning from Experience in Minecraft

We apply two learning procedures to the traces of one or more of the training procedures. The **frequentist** procedure applies simple statistical sampling from random, ordered, or expert traces to select the best choice. The frequentist procedure leverages all state knowledge even though some state may be irrelevant to decision making. To apply these traces, we then collated the results into tables that counted the subgoal chosen for a particular state where, if the subgoal was successful, we add 3 to the frequency, otherwise we subtract 1. The frequentist procedure then chooses the subgoal with the highest frequency based on the current state. If the GRPROCESS encounters a state that was not observed in any of its training traces, it fails to reach the goal location.

The **decision tree** (d-tree) procedure learns a decision tree over past experience to select the best subgoal. We used the J48 algorithm implemented in WEKA[6]. Figure 7 shows the tree learned from expert traces. The world object at that position is indicated by a single letter: cobblestone (C), emerald (E), air (A), lava (L), water (W), and gold (G). Thus the first state listed in the tree is the block immediately in front of Steve's head: left/right of 0, front/back of 1, and height of 1.

Evaluation We measured the number of subgoal choices to complete each variant using each of the three training procedures. We also counted the number of failed attempts the GRPROCESS tried before reaching the goal location or failing. We ran 10 trials for each procedure in each variant on random course lengths of 5, 10, 15, and 20 sections. The experiment times out if time exceeds 240 steps or when, during the frequentist procedure, an unknown state is encountered.

The expert and learned approaches never fail, so we focus our discussion on the runtime, which is a proxy for the number of steps taken. We measured the run time taken for the GRPROCESS to complete the obstacle course using each of the three procedures. We expected the expert procedure to have the lowest elapsed time, the random procedure to have the highest elapsed time, and ordered to be between random

[6] http://www.cs.waikato.ac.nz/~ml/weka/

```
lp0fp1hp1 = C
|   lp0fp1hp2 = C: RemoveObstacle (43.0)
|   lp0fp1hp2 = A: CreateStairs (19.0)
|   lp0fp1hp2 = E: RemoveObstacle (0.0)
|   lp0fp1hp2 = L: RemoveObstacle (0.0)
|   lp0fp1hp2 = W: RemoveObstacle (0.0)
|   lp0fp1hp2 = G: RemoveObstacle (0.0)
lp0fp1hp1 = A
|   lp0fp1hn1 = C: WalkTo (731.0)
|   lp0fp1hn1 = A: WalkTo (53.0)
|   lp0fp1hn1 = E: WalkTo (0.0)
|   lp0fp1hn1 = L: CreateBridge (42.0)
|   lp0fp1hn1 = W: CreateBridge (36.0)
|   lp0fp1hn1 = G: WalkTo (17.0)
lp0fp1hp1 = E: WalkTo (0.0)
lp0fp1hp1 = L: WalkTo (0.0)
lp0fp1hp1 = W: WalkTo (0.0)
lp0fp1hp1 = G: WalkTo (0.0)
```

Figure 7: The decision tree learned from the expert traces.

and expert. This is because the expert procedure checks the state of the environment before choosing a sub-goal, and therefore makes an informed decision. In the random and ordered procedures, the GRPROCESS relies on a random or pre-set order of sub-goals to choose from, which could lead to inefficiencies if a sub-goal is not appropriate, but still achieved. When applying learning, we expected the frequentist procedure to have an elapsed time between ordered and expert and we expected to see an effect of using different traces.

Results Table 1 shows the elapsed time as the number of sections in the obstacle course increase from 5 to 20. The left-most column indicates one or more training traces used by the learning mechanism: random (R), ordered (O), or expert (E). The number of samples is too low for statistical testing, but we plan to run a full experiment and report such testing in future revisions.

The top subtable, Training, shows the average runtime during trace collection; a dash indicates a time out and failure to reach the gold block for all runs. On average, expert performs best, random worst, and ordered in the middle. However, in looking more closely at the runs, we noted that the expert procedure does not always outperform ordered. This finding does not meet our expectation that the expert will dominate, but the results are strongly suggestive that it generally performs best. We note that the ordered procedure does not perform that much worse than the expert, which is a theme we return to several times.

We counted the number of failed subgoal attempts, which indicates how often a procedure selects an inappropriate subgoal that the Controller disallowed. We only discuss these results and do not show the data. The expert procedure never fails a subgoal. The random procedure had the most number of failed goal attempts, and the ordered procedure was in the middle of the other two. These trends were expected as the random and ordered procedures do not check the environment state before making an informed decision, and show that the expert procedure is better in terms of choosing the right sub-goals.

	5		10		15		20	
Train	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ
Training								
R	-	-	-	-	-	-	-	-
O	21.5	1.7	41.8	3.3	56.8	3.3	79.8	2.1
E	16.3	1.3	23.8	1.3	46.0	4.1	64.8	5.0
Frequentist								
R	-	-	-	-	-	-	-	-
O	17.5	1.7	33.3	2.1	48.8	5.1	63.5	3.5
RO	17.5	1.7	33.3	2.1	48.8	5.1	63.5	3.5
E	16.3	1.3	32.8	1.3	47.3	5.1	65.0	4.6
RE	16.3	1.3	32.8	1.3	46.7	6.1	65.0	4.6
OE	17.5	1.7	33.3	2.1	48.8	5.1	63.5	3.5
ROE	17.5	1.7	33.3	2.1	49.3	5.4	63.5	3.5
Decision Tree								
R	-	-	-	-	-	-	-	-
O	6.7	0.6	12.0	0.8	18.3	2.1	23.5	2.1
RO	6.7	0.6	12.0	0.8	18.0	2.8	23.0	0.0
E	5.8	1.0	13.3	0.5	18.3	2.4	27.0	1.2
RE	5.8	1.0	13.3	0.5	18.3	2.4	27.0	1.2
OE	6.7	0.6	12.0	0.8	18.3	2.1	23.0	1.7
ROE	6.7	0.6	12.0	0.8	18.3	2.1	23.3	1.5

Table 1: Mean runtime and standard deviation for the study.

Table 1 (middle) shows the results obtained from the frequentist procedure learning using various combinations of the training traces. Using either expert traces (E row) or ordered traces (O row) only resulted in substantially similar runtimes to the original expert traces, suggesting that either kind of trace is suitable for biasing learning. Combining the two traces (OE) did not appear to change the behavior.

Using only random traces (R) did not produce an effective policy. But it also does not appear that adding the random trace to expert (RE), ordered (RO), or their both (ROE) cause a significant degradation of the results.

Table 1 (bottom) shows the results obtained from decision tree learning. We can observe a dramatic improvement in the runtime with this procedure. Moreover, the ordered tree (O) sometimes has better average performance than the expert tree (E). Adding random traces (RO, RE, ROE) did not appear to diminish the results substantially. As seen in Figure 6, we use 15 states per observation. The ordered tree (O) makes effective decisions using between 3 and 5 states, while the expert tree (E) makes effective decisions examining at most 2 states. Clearly, there is great benefit to learning which observations matter for effective decision making.

7 Other ACTORSIM Connectors

Other projects apply, extend, or propose ACTORSIM to work with additional simulators. We present a snapshot of each project to highlight how ACTORSIM assists in studying goal reasoning.

Foreign Disaster Relief The longest-running project for ACTORSIM is Foreign Disaster Relief, where we have studied

how to perform goal reasoning to coordinate teams of robotic vehicles (Roberts et al. 2015a), estimating high-fidelity simulations using a faster, but lower-fidelity estimates, and its application to play-calling (Apker et al., 2015). The most recent extension of this work has extended Goal Reasoning with Information Metrics (GRIM) by Johnson et al. (2016). The ACTORSIM codebase, in particular the Goal Reasoning Library, had its genesis in abstractions developed during this project. Similar to the studies presented, ACTORSIM uses the MASON simulator for the scenarios of this project. However, the set of motion and path planning primitives is simplified in that it does not leverage the LTL templates or vehicle controllers mentioned in Roberts et al. (2015a).

StarCraft StarCraft:Brood War is a Real Time Strategy (RTS) game developed by Blizzard Entertainment. At an abstract level, it is an economic and military simulation. Players build an economy to gather resources, use these resources to train an army, then use this army to attempt to defeat their opponent, either in direct engagements or through disrupting their economy. It has a number of desirable properties as an artificial intelligence testbed, and has seen a good deal of research in recent years (Ontanon et al. 2013).

ACTORSIM integrates with an existing game agent developed by Churchill et al., UAlbertaBot (UAB)[7]. UAB interfaces directly with the game of Brood War using the Brood War API (BWAPI)[8], through which it can issue commands to units and monitor the observable state of the game. It is a modular agent on which researchers can build their systems.

The ACTORSIM Connector controls a subset of the behavior of the agent, letting UAB control the remainder. For example, if ACTORSIM creates a goal to attack a specific region of the map, UAB will decide the formation and specific unit commands necessary to achieve that goal. The behavior controlled by ACTORSIM is currently region-level positioning, soon to include economic growth decisions.

We have used ACTORSIM to emulate the original hand-coded behaviors of UAB, and are in the process of implementing more complex goals to demonstrate the additional expressivity of the agent using our system. In addition, we are working on automatically learning the EVALUATEfunction based on replays of professional human players, which are available online in large quantities.

8 Related Work

Researchers have applied goal reasoning to other domains, such as the Tactical Action Officer (TAO) Sandbox (Molineaux et. al., 2010). Using the Autonomous Response to Unexpected Events (ARTUE) agent, they implemented goal-driven autonomy; ARTUE can reason about what goals to achieve based on the changing environment, in this case a strategy simulation for TAOs to train in anti-submarine warfare. Goal reasoning has been used in other gaming domains such as Battle of Survival, a real-time strategy game (Klenk et al., 2013).

Goal refinement builds on the work in plan refinement (Kambhampati, Knoblock, & Yang 1995), which equates different kinds of planning algorithms in plan-space and state-space planning. Extensions incorporated other forms of planning and clarify issues in the Modal Truth Criterion (Kambhampati and Nau 1994). More recent formalisms such as Angelic Hierarchical Plans (Marthi et al. 2008) and Hierarchical Goal Networks (Shivashankar et al. 2013) can also be viewed as leveraging plan refinement. The focus on constraints in plan refinement allows a natural extension to the many integrated planning and scheduling systems that use constraints for temporal and resource reasoning.

The goal lifecycle bears close resemblance to that of Harland et al (2014) and earlier work (Thangarajah et al. 2010). They present a goal lifecycle for BDI agents, provide operational semantics for their lifecycle, and demonstrate the lifecycle on a Mars rover scenario. Recently, Cox et al. (2016) proposed a model for goal reasoning based on planning. We hope to characterize the distinction between these models in future work.

9 Summary and Future Work

In this paper, we formalized a semantics for goal reasoning and applied our implementation, called ACTORSIM, to a pilot study in Minecraft. For the task that we examined, we developed three methods to selection sub-goals: random, ordered, and expert. Using the results of these methods, we examined two learning methods. We showed that the expert selection is the most efficient based on the elapsed time and the number of failed goal attempts, and that the random selection was the least efficient followed by the ordered selection. For the frequentist approach, we showed that the expert and ordered traces yielded the best performance, and that adding random traces did not seem to cause too much harm.

In the future, we plan to incorporate more complex tasks such as having an agent protect itself against creepers and mobs. A first step in this direction will be to encode our goal network using the goal lifecycle provided in ACTORSIM, since our current implementation applies goal reasoning without using much of its functionality. This will allow us to build (or learn) more sophisticated goal networks and to take advantage of existing planning and scheduling techniques in ACTORSIM. Finally, we plan to include non-playing characters in Minecraft with our resulting goal networks.

Broader dissemination of ACTORSIM will foster deeper study and enriched collaboration between researchers interested in goal reasoning, planning, and acting. ACTORSIM complements existing open source planning systems with a standardized implementation of goal reasoning so researchers can focus on (1) designing goals and goal transitions for their system (2) linking ACTORSIM to their particular simulator, and (3) studying goals and behavior in the dynamic environment provided by the simulator. By releasing it as an open source package, we lay a foundation for advanced studies in goal reasoning that include integration with additional simulators and planning systems, formal models, and empirical studies that examine decision making in challenging, dynamic environments.

Our upcoming projects include extending ACTORSIM to actual robotic systems that include the Roomba system

[7]http://www.github.com/davechurchill/ualbertabot

[8]http://www.github.com/bwapi/bwapi

and a set of Hubo's. The architecture of ACTORSIM is now developed enough that we can also start to consider integrating it with more sophisticated robot simulators such as Gazebo or the the robocup simulators. We believe this area presents a great deal of promise for the formal model of goal reasoning we present in this paper as well as for ACTORSIM.

Acknowledgments

This research was funded by OSD and NRL. Ron Alford performed part of this work under an ASEE postdoctoral fellowship at NRL. We thank the anonymous reviewers for comments that helped improve the paper.

References

Abel, D.; Hershkowitz, D. E.; Barth-Maron, G.; Brawner, S.; OFarrell, K.; MacGlashan, J.; and Tellex, S. 2015. Goal-based action priors. In *Proc. Int'l Conf. on Automated Planning and Scheduling*.

Alford, R.; Shivashankar, V.; Roberts, M.; Frank, J.; and Aha, D. W. to appear. Hierarchical planning: Relating task and goal decomposition with task sharing. In *Proc. of the Int'l Joint Conf. on AI (IJCAI)*. AAAI Press.

Aluru, K.; Tellex, S.; Oberlin, J.; and Macglashan, J. 2015. Minecraft as an experimental world for AI in robotics. In *AAAI Fall Symposium*.

Apker, T.; Johnson, B.; and Humphrey, L. 2016. Ltl templates for play-calling supervisory control. In *Proc. AIAA @Infospace*.

Cox, M. T.; Alavi, Z.; Dannenhauer, D.; Eyorokon, V.; Munoz-Avila, H.; and Perlis, D. 2016. MIDCA: A metacognitive, integrated dual-cycle architecture for self-regulated autonomy. In *AAAI*.

Dannenhauer, D., and Munoz-Avila, H. 2015. Raising expectations in gda agents acting in dynamic environments. In *Proc. of the Int'l Joint Conf. on AI (IJCAI)*. AAAI Press.

Geier, T., and Bercher, P. 2011. On the decidability of HTN planning with task insertion. In *Proc. of the 22nd Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 1955–1961. AAAI Press.

Ghallab, M.; Nau, D.; and Traverso, P. 2014. The actor's view of automated planning and acting: a position paper. *Artificial Intelligence* 208:1–17.

Giacomo, G. D.; Gerevini, A. E.; Patrizi, F.; Saetti, A.; and Sardina, S. 2016. Agent planning programs. *Artificial Intelligence* 231:64–106.

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

Johnson, B.; Roberts, M.; Apker, T.; and Aha, D. to appear. Goal reasoning with information measures. In *Proceedings of the Conf. on Advances in Cognitive Systems*.

Klenk, M.; Molineaux, M.; and Aha, D. 2013. Goal-driven autonomy for responding to unexpected events in strategy simulations. *Computational Intelligence* 29(2):187–206.

Molineaux, M., and Aha, D. W. 2014. Learning unknown event models. In *AAAI*.

Munoz-Avila, H.; Aha, D.; Jaidee, U.; Klenk, M.; and Molineaux, M. 2010. Applying goal directed autonomy to a team shooter game. In *FLAIRS*, 465–470.

Munoz-Avila, H.; Wilson, M. A.; and Aha, D. W. 2015. Guiding the ass with goal motivation weights. In *2015 Annual Conference on Advances in Cognitive Systems: Workshop on Goal Reasoning*.

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

Nau, D. S.; Ghallab, M.; and Traverso, P. 2015. Blended planning and acting: Preliminary approach, research challenges. In *AAAI Conf. on Artificial Intelligence*.

Nau, D. 2007. Current trends in automated planning. *Art. Intell. Mag.* 28(40):43–58.

Ontanon, S.; Synnaeve, G.; Uriarte, A.; Richoux, F.; Churchill, D.; and Preuss, M. 2013. A survey of real-time strategy game ai research and competition in starcraft. *IEEE Trans. Comput. Intellig. and AI in Games* 5:293–311.

Roberts, M.; Apker, T.; Johnston, B.; Auslander, B.; Wellman, B.; and Aha, D. W. 2015a. Coordinating robot teams for disaster relief. In *International Conference of the Florida Artificial Intelligence Research Society*.

Roberts, M.; Vattam, S.; Alford, R.; Auslander, B.; Apker, T.; Johnson, B.; and Aha, D. W. 2015b. Goal reasoning to coordinate teams for disaster relief. In *Working Notes of the PlanRob Workshop at ICAPS*.

Roberts, M.; Alford, R.; Shivashankar, V.; Leece, M.; Gupta, S.; and Aha, D. to appear. Goal reasoning, planning, and acting with ActorSim, the Actor Simulator. In *Proceedings of the Conf. on Advances in Cognitive Systems*.

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. Int. Foundation for AAMAS.

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*.

Vattam, S., and Aha, D. W. 2015. Case-based plan recognition under imperfect observability. In *ICCBR*.

Vattam, S.; Klenk, M.; Molineaux, M.; and Aha, D. 2013. Breadth of approaches to goal reasoning: A research survey. In Aha, D.; Cox, M.; and Muoz-Avila, H., eds., *Goal Reasoning: Papers from the ACS Workshop (Technical Report CS-TR-5029). College Park, MD: University of Maryland, Department of Computer Science*, 222–231.

Wilson, M. A.; Molineaux, M.; and Aha, D. W. 2013. Domain-independent heuristics for goal formulation. In *FLAIRS*.

Young, J., and Hawes, N. 2012. Evolutionary learning of goal priorities in a real-time strategy game. In *Proc. of the AIIDE*. AAAI Press.