Advances in Cognitive Systems 4 (2016) Submitted 4/2016; published 6/2016

Goal Reasoning, Planning, and Acting with ACTORSIM, The Actor Simulator[1]

Mark Roberts[1] MARK.ROBERTS.CTR@NRL.NAVY.MIL
Vikas Shivashankar[2] VIKAS.SHIVASHAKAR@KNEXUSRESEARCH.COM
Ron Alford[3] RALFORD@MITRE.ORG
Michael Leece[4] MLEECE@SOE.UCSC.EDU
Shubham Gupta[5]
David W. Aha[6] DAVID.AHA@NRL.NAVY.MIL
[1]NRC Postdoctoral Fellow; Naval Research Laboratory (Code 5514); Washington, DC
[2]Knexus Research Corp.; Springfield, VA
[3]MITRE; McLean, VA
[4]Dept. of Computer Science; Univ. of California Santa Cruz; Santa Cruz, CA
[5]Thomas Jefferson High School for Science and Technology; Alexandria, VA
[6]Naval Research Laboratory (Code 5514); Washington, DC

Abstract

Goal reasoning is maturing as a field, but it lacks a model with clear semantics in a readily available implementation that researchers can build upon. This paper presents contributions that address this gap. First, we formalize goal reasoning with crisp semantics by extending a recent formalism called goal-task network planning. Second, we describe an open source package, called ACTORSIM, that partially implements the semantics of the formal model. Finally, we use ActorSim in a study to examine whether a machine learning technique can improve subgoal selection using goal reasoning in the game of Minecraft. The study reveals that simple mechanisms for gathering experience improve over less knowledge intensive or random approaches for the domain we study.

1. Introduction

Goals are a unifying structure for designing and studying intelligent 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 an actor 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 which goal(s) to progress given trade-offs in dynamic, possibly adversarial, environments. Thus, goal reasoning is a critical component for enabling more responsive and capable autonomy.

1. This paper supersedes work by Roberts et al. (2016) presented at the Planning and Robotics Workshop at ICAPS-2016.

© 2016 Cognitive Systems Foundation. All rights reserved.

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, & Aha, 2013), goal motivators (Munoz-Avila, Wilson, & Aha, 2015), goal recognition (Vattam & Aha, 2015), goal prioritization (Young & Hawes, 2012), explanation generation (Molineaux & 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.

To this end, we describe the Actor Simulator, ACTORSIM, as a general platform for conducting studies of goal reasoning in simulated environments. We draw inspiration from the literature in planning, where 15 years of International Planning Competitions has blossomed into a research ecosystem of nearly 100 open source planning systems and hundreds of planning benchmarks[3] in a standardized language called the Planning Domain Definition Language (Gerevini et al., 2008). Similarly, we aim to foster studies of goal reasoning. After discussing some background (§2) we present contributions that include:

A **formal model** of goal reasoning and its semantics (§3), extending previous work by Roberts et al. (2015b) and building on a hybrid model of planning called Goal-Task Network (GTN) planning (Alford et al., 2016), 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, which is critical in a system that performs goal reasoning and deliberation as discussed by Ghallab, Nau, & Traverso (2014).

An **open source platform** called ACTORSIM[4], that partially implements this formal model (§4) ACTORSIM's initial design began with 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 tasks defined in Minecraft (§5) with preliminary results showing that learning from structured experience to select subgoals improves behavior for a simple navigation task, expert knowledge is useful but not essential for effective decision making in this task, and costly random knowledge gathering is ineffectual. Our results complement existing studies on gathering and learning from experience demonstrating that goal reasoning can overcome some limitations of action selection approaches.

2. The latest workshop is described at http://makro.ink/ijcai2016grw/
3. See the latest summary by Valatti et al. (2015) or previous competitions at http://ipc.icaps-conference.org/
4. Available at http://makro.ink/actorsim

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 (left)). 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.

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 et al. (2014). GTN planning augments the notation of Geier & 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

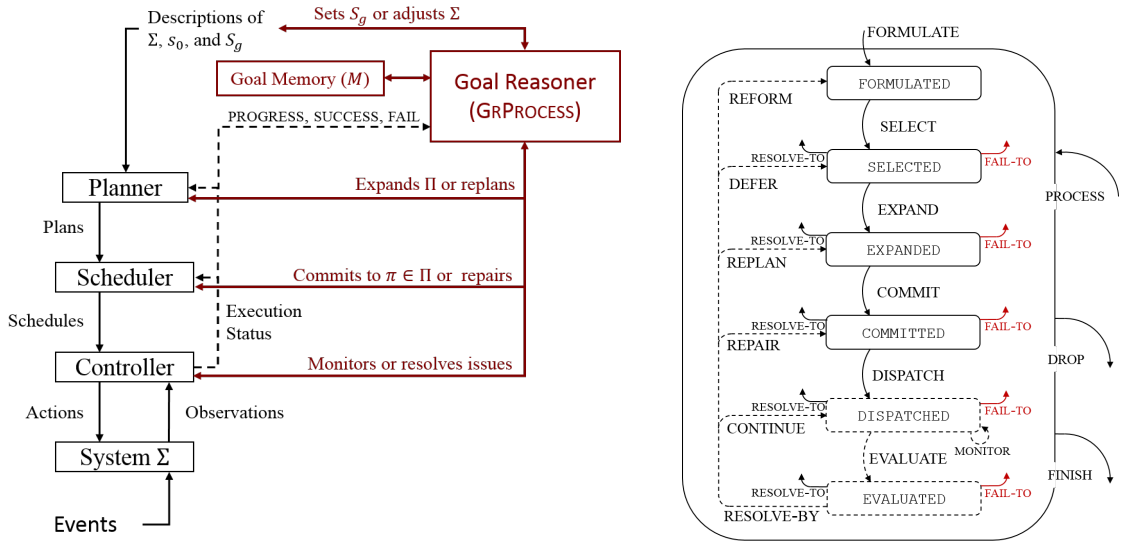

Figure 1. Relating goal reasoning with online planning (left), where the GRPROCESS works with a goal memory to monitor and modify the goals or planning model of the system. The goal memory stores goal nodes that transition according to the goal lifecycle (right), where refinement strategies (arcs) denote decision points of the GRPROCESSand modes (rounded boxes) denote the status of a goal in the goal memory.

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 $\gamma : 2^{\mathcal{L}} \times \mathcal{O} \to 2^{\mathcal{L}}$ 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 1 (right) 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., $\mathcal{N}.N \coloneqq (s, gtn')$ 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

Figure 1 (right) 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.

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).

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: $\mathcal{N}'o \coloneqq$ `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 $\mathcal{L}$ is a propositional language, $\mathcal{O}$ and $\mathcal{M}$ are defined as in §2.1, R_d is the default set of strategies defined next, R_p is a set of strategies provided by a domain designer, and M_0 is the initial goal memory. P_{gr} is solvable iff there is a progression $M_0 \rightarrow^*_R M_k$, where $M_k = \emptyset$. Recall that we consider only achievement goals in this paper, so this definition of a solution is sufficient. However, a more complete goal taxonomy will require an extended definition for valid solutions.

4. The Actor Simulator

The Actor Simulator, ACTORSIM (Figure 2), implements the goal lifecycle of §3.2. Its initial design grew from work on robotic applications to Foreign Disaster Relief operations (Roberts et al., 2015b) and has since been extended to several other domains. It complements existing open source planning systems with a standardized implementation of goal reasoning and also provides links to simulators that can simulate multiple agents interacting within a dynamic environment. The **Goal Refinement Library** is a standalone library that provides goal management via the goal memories and the data structures for transitioning goals throughout the system via the goal lifecycle. It contains

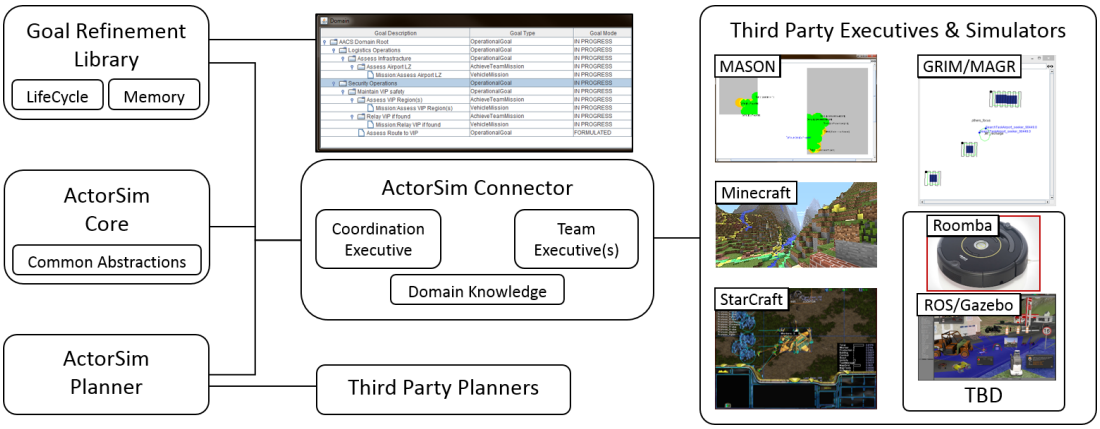

Figure 2. The Component Architecture of ACTORSIM .

the default implementations for goals, goal types, goal refinement strategies, the goal memory, domain loading, and domain design. This library includes a simple display to show the state of goal-task networks within the memory. The **Core** provides the interfaces and minimal implementations of the platform. It contains the essential abstractions that apply across many simulators and domains. This component contains information about Areas, Locations, Actors, Vehicles, Symbols, Maps, Sensors, and configuration details. The **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. The **Connector** manages the link between the goal memory, domain knowledge and the executives running on existing simulators. This component contains abstractions for Tasks, Events, Human interface Interaction, Executives (i.e., Controllers), and event notification. A variety of **Executives and Simulators** can be connected to ACTORSIM via a direct library link in Java or through Google's protocol buffers[5]. Currently supported simulators include George Mason University's MASON[6] 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 plan to eventually link ACTORSIM to physical hardware.

5. Overcoming Obstacles in Minecraft

Researchers have recently used the Minecraft game for the study of intelligent agents (Aluru et al., 2015) . In this game, a human player moves a character, Steve, to explore a 3D voxel world while gathering resources and surviving dangers. Steve's limited inventory can hold resources (e.g., sand, wood, stone) used to craft into new items that can in turn be used to construct tools (e.g., a pickaxe

5. https://developers.google.com/protocol-buffers/
6. http://cs.gmu.edu/~eclab/projects/mason/

for mining) or structures (e.g., a shelter or lookout tower). Certain blocks (e.g., lava, deep water), hostile characters (e.g., creepers), or falling more that two blocks can damage Steve's health.

We focus on the problem of having the GRPROCESS move Steve to a goal by navigating through a course in a much simpler subset of the world. In previous work, researchers developed a learning architecture to interact with Minecraft (Abel et al., 2015) . Their method allows the virtual player to disregard unneeded actions for navigating the maze. The set of possible choices available to achieve even this *simple* goal is staggering; for navigating a 15x15 maze in Minecraft, Abel et al. (2015) estimate the state space to be nearly one million states.

Our technical approach differs from prior research in that we aim to develop a GRPROCESS that uses increasingly sophisticated goal-task networks and learned experience about when to apply them. The ACTORSIM-Minecraft connector leverages a reverse-engineered game plugin called the Minecraft Forge API (Forge), which provides methods for manipulating Minecraft. We implemented axis-aligned motion primitives such as looking, moving, jumping, and placing or destroying blocks. For this paper, Steve always faces North with the course constructed to the North and Steve interacts with a limited set of world objects: cobblestone, dirt, clay, air, lava, water, emerald, and gold.

The GRPROCESS can observe blocks directly around Steve's avatar and we use an alpha-numeric code to indicate the local position relative to Steve's feet. A relative position is labeled with a unique string relative to Steve "[lN | rN][fN | bN][uN | dN]" when N is a non-negative integer and each letter designates **l**eft,**r**ight, **f**ront, **b**ack, **u**p, and **d**own. Thus, "f1d1" indicates the block directly in front and down one (i.e., the block that Steve would stand on after stepping forward one block), while "l1" indicates the block directly to the left at the same level as Steve's feet. We abstract the world object at that local position by a single letter: air (A), safe or solid (S), water (W), or lava (L).

The GRPROCESS can move using one of five subgoals: stepTo, stepAround, build a bridge, build a stair one block high, and mine. The Controller ensures that Steve will not violate safety constraints by falling too far or walking into a pool of lava or water (the character does not currently swim), but the GRPROCESS must learn these constraints independently. For example, if the subgoal to step forward is selected when lava is directly in front of Steve the Controller ignores the move, resulting in failure that requires additional reasoning. Thus, subgoals do not contain operational knowledge and the GRPROCESS must learn to select them based on the current state.

Our task-goal representation complements 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 hidden so the GRPROCESS cannot perform offline planning; there must be an interleaving of perception, goal reasoning, and acting. Third, the operational semantics of committing to a subgoal are left to the Controller. Although random exploration is possible, we will present evidence that that such an approach is untenable, corroborating the findings of Abel et al. (2015) that the state/action space is too large to explore without a bias.

Learning from Experience *Our research hypothesis is that making effective choices at the* GTN *level can be done by learning from execution traces that lead to getting to the goal in fewer steps or failing less frequently.* We examine what kind of experience is most valuable. To this end, we describe a pilot study that leverages prior experience to learn a subgoal selection policy. Figure 3

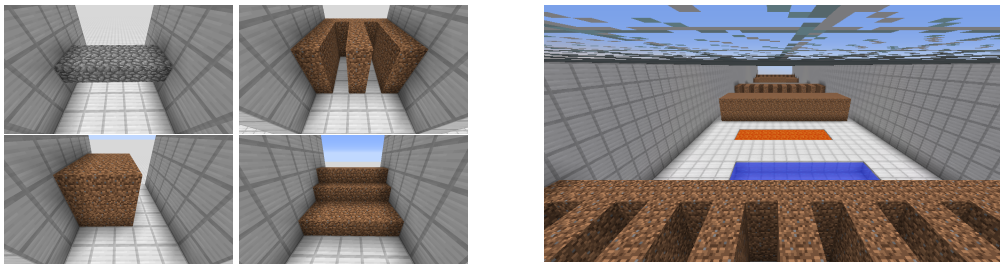

Figure 3. Left: four example section types (top left to bottom right) include arch, comb, pillar, and steps. Right: portion of a course where the GRPROCESS must traverse from the emerald block behind it (not shown) to a gold block in front of it (not shown). Glass blocks on the top prevent Steve from walking along the wall.

(left) shows the four of the ten section types the GRPROCESS may encounter: arch, comb, pillar, and hill; not shown are lava, short wall (2 blocks high), tall wall (3 blocks high), deep pond (water 3 deep), pond (water 2 deep), and swamp (water 1 deep). Figure 3 (right) shows a portion of a course with upcoming sections of a comb, swamp, lava and short wall. Each obstacle has an appropriate subgoal choice. For lava or ponds, the best choice is to create a bridge if approaching the center or to go around if approaching the edge. For the short walls, the best subgoal is to create a single stair and step up. For the tall walls, combs, and pillars, the best subgoal is to mine through if approaching the center or go around if approaching the edge.

We collect three kinds of traces for choosing the five subgoals that vary in how much state they consider. **Random** ignores state and selects a subgoal with uniform probability. **Ordered** also ignores state and selects the subgoals in the same order: stepAround, stepTo, bridge, mine, and stairs. If the Controller allows the move it is taken and ordered selection restarts from the beginning of the order, otherwise the next subgoal is attempted. If all subgoals fail then the run terminates. **Expert** examines the full state and is hand-coded (by an author of this paper) to select the best subgoal. The Ordered and Expert procedures may be randomized, in which case we will call the trace ExpertNN or OrderedNN for the ratio of how often a random choice is made instead of the expert choice. For example Expert10 indicates that 10% of the time a random choice is selected. For each trace, we capture the state, distance to the goal, subgoal chosen, whether the chosen subgoal succeeded, and how many blocks were placed or mined. The Random procedure and randomized variants can fail to reach the gold. The Expert and Ordered procedures rarely fail to reach the gold but both represent biased knowledge or sampling of the state or choices.

To learn from these traces we create a *decision tree (d-tree)* using the J48 algorithm implemented in WEKA[7]. Figure 4 shows two tree learned from Expert traces. The leaves of a tree represent the command to be taken and inner nodes represent (location, block) values. For example, the first two lines indicate that if the blocks directly in front of the player above and below the foot level are air, then the character is to perform the stepTo command.

7. `http://www.cs.waikato.ac.nz/~ml/weka/`

```
f1u1 = A
|   f1d1 = A: DoStepTo (275.0)
|   f1d1 = S
|   |   f2d1 = A: DoStepTo (174.0)
|   |   f2d1 = S: DoStepTo (3591.0/22.0)
|   |   f2d1 = W
|   |   |   f2r2d1 = S: DoStepAroundTo (18.0)
|   |   |   f2r2d1 = W
|   |   |   |   f2l2d1 = S: DoStepAroundTo (12.0/2.0)
|   |   |   |   f2l2d1 = W: DoStepTo (343.0)
|   |   f2d1 = L
|   |   |   f2r2d1 = S: DoStepAroundTo (4.0)
|   |   |   f2r2d1 = L
|   |   |   |   f2l2d1 = S: DoStepAroundTo (3.0)
|   |   |   |   f2l2d1 = L: DoStepTo (142.0)
|   f1d1 = W
|   |   f1r1d1 = S: DoStepAroundTo (11.0)
|   |   f1r1d1 = W
|   |   |   f1l1d1 = S: DoStepAroundTo (6.0)
|   |   |   f1l1d1 = W: DoBuildBridgeTo (345.0)
|   f1d1 = L
|   |   f1r1d1 = S: DoStepAroundTo (4.0)
|   |   f1r1d1 = L
|   |   |   f1l1d1 = S: DoStepAroundTo (3.0)
|   |   |   f1l1d1 = L: DoBuildBridgeTo (142.0)
f1u1 = S
|   f1u2 = A
|   |   f2u1 = A: DoMineTo (25.0)
|   |   f2u1 = S: DoBuildStairsTo (143.0)
|   f1u2 = S
|   |   f1l1 = A: DoStepAroundTo (6.0)
|   |   f1l1 = S
|   |   |   f1r1 = A: DoStepAroundTo (2.0)
|   |   |   f1r1 = S: DoMineTo (444.0)
```

```
f1u1 = A
|   f1d1 = A: DoStepTo (135.0/13.0)
|   f1d1 = S
|   |   f2d1 = A: DoStepTo (49.0/5.0)
|   |   f2d1 = S: DoStepTo (1078.0/125.0)
|   |   f2d1 = W
|   |   |   f2l2d1 = S: DoStepAroundTo (4.0)
|   |   |   f2l2d1 = W
|   |   |   |   f2r2d1 = S: DoStepAroundTo (3.0)
|   |   |   |   f2r2d1 = W: DoStepTo (67.0/6.0)
|   |   f2d1 = L
|   |   |   f1r2d1 = S
|   |   |   |   f2l2d1 = S: DoStepAroundTo (7.0/1.0)
|   |   |   |   f2l2d1 = L: DoStepTo (7.0/2.0)
|   |   |   f1r2d1 = L: DoStepTo (14.0/1.0)
|   |   f2d1 = U: DoStepTo (0.0)
|   f1d1 = W: DoBuildBridgeTo (67.0/5.0)
|   f1d1 = L
|   |   f1l1d1 = S
|   |   |   f1l1 = A: DoStepAroundTo (4.0)
|   |   |   f1l1 = S: DoBuildBridgeTo (5.0)
|   |   f1l1d1 = L: DoBuildBridgeTo (15.0/4.0)
|   f1d1 = U: DoStepTo (0.0)
f1u1 = S
|   f1u2 = A
|   |   f2u1 = S: DoBuildStairsTo (70.0/15.0)
|   f1u2 = S: DoMineTo (129.0/13.0)
```

Figure 4. The decision tree learned from 30 traces of Expert (left) and Expert25 (right). The number in parentheses indicates how frequently that combination was observed.

Results We randomly generated 30 courses with 20 sections each; all procedures use these same 30 courses. A run terminates when Steve reaches the goal, when 20 subgoals in a row fail for Random and Expert, or when no subgoal succeeds for Ordered. Table 1 (top) shows the results for the training traces. The Expert procedure is successful at completing all courses with only five failed subgoal attempts. The Random procedure never finishes a course and has nearly twice as many failures. One reason for this is that the stepTo subgoal is much more frequently chosen than the other subgoals so random choice is too unbiased. The biased sampling of the Ordered procedure performs almost identically to Expert. These results suggest not only that completely random exploration is unjustified but also that the effort to hand-code a custom controller may not have been warranted. Although the Ordered procedure has 5! possible orderings, learning a biased sampling may provide sufficient knowledge for effective decision making. We will examine such an approach in future work.

Randomizing the Expert procedure dramatically reduces its effectiveness as Table 1 (middle) shows. Even 10% random choice a significant drop in course completion is observed. At 25%, the failure rate is equal to purely random choice.

Table 1 (bottom) shows the benefit of learning from the Ordered or Expert traces. The decision tree learned from the expert traces performs very similar to the original Expert procedure, but the higher number of placed and mined blocks indicate a bias toward mining and bridges over walking around obstacles. The tree learned from Random traces fails to produce an effective policy. For

	Course		Subgoals			Blocks	
Method	N	Completed	Attempts	Success	Fail	Placed	Mined
Expert	30	30	5698	5693	5	630	938
Ordered	30	30	5697	5693	4	630	938
Random	30	0	14847	5429	9418	1428	1970
Randomized Variants							
Expert-05	30	13	3880	3679	201	449	576
Expert-10	30	8	3730	3374	356	430	556
Expert-25	30	0	2144	1664	480	228	298
Decision Tree							
Random	30	0	-	-	-	-	-
Ordered	30	30	5721	5712	9	637	996
Expert	30	30	5716	5709	7	637	996
Expert-05	30	30	5730	5720	10	637	1030
Expert-10	30	30	5762	5747	15	666	1030
Expert-25	30	30	5800	5795	5	670	1104

Table 1. Results for various procedures used in this study.

the random variants of Expert, which perform poorly on their own, the decision tree overcomes deficiencies and performs similarly to the decision trees trained with Expert traces.

6. Other ACTORSIM Connectors

As shown in Figure 2, ACTORSIM is integrated 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, Apker, Johnston, Auslander, Wellman, & Aha, 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 (Ontanon et al., 2013).

ACTORSIM uses a protobuf interface to integrate with an existing game agent developed by David Churchill called UAlbertaBot (UAB)[8]. UAB interfaces directly with the game of Brood War using the Brood War API (BWAPI)[9], 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 EVALUATE function based on replays of professional human players, which are available online in large quantities.

7. 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 TAO's 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). While our approach also applies goal reasoning, we present the first use of a GTN to this purpose in a game environment.

Goal refinement builds on the work in plan refinement (Kambhampati et al., 1995), which equates different kinds of planning algorithms in plan-space and state-space planning. More recent formalisms such as Angelic Hierarchical Plans (Marthi et al., 2008) and Hierarchical Goal Networks (Shivashankar et al., 2012) 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 by 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.

8. Summary

We presented a formal semantics for goal reasoning and applied our partial implementation of those semantics, called ACTORSIM, to a pilot study in Minecraft. In this study we trained a decision using traces from Random, Ordered, and Expert procedures. We showed that, for this limited domain,

8. http://www.github.com/davechurchill/ualbertabot
9. http://www.github.com/bwapi/bwapi

learning from structured exploration (i.e., the Ordered traces) is as effective as Expert exploration and costly random knowledge gathering is ineffectual. In the future, we will incorporate more complex goal-task networks that solve even more complicated tasks.

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 provide 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.

Acknowledgements

This research was funded by OSD and NRL. We thank the anonymous reviewers for their comments that helped improve this paper.

References

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

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

Aluru, K., Tellex, S., Oberlin, J., & Macglashan, J. (2015). Minecraft as an experimental world for AI in robotics. *AAAI Fall Symposium*.

Apker, T., Johnson, B., & Humphrey, L. (2016). Ltl templates for play-calling supervisory control. *Proc. AIAA @Infospace*.

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

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

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

Gerevini, A. E., Haslum, P., Long, D., Saetti, A., & Dimopoulos, Y. (2009). Deterministic planning in the fifth international planning competition: Pddl3 and experimental evaluation of the planners. *Artificial Intelligence*, *173*, 619–668.

Ghallab, M., Nau, D., & 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., & Sardina, S. (2016). Agent planning programs. *Artificial Intelligence*, *231*, 64–106.

Harland, J., Morley, D. N., Thangarajah, J., & 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., & Aha, D. (to appear). Goal reasoning with information measures. *Proceedings of the Conf. on Advances in Cognitive Systems*.

Kambhampati, S., Knoblock, C. A., & Yang, Q. (1995). Planning as refinement search: A unified framework for evaluating design tradeoffs in partial-order planning. *Artificial Intelligence*, *76*, 168–238.

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

Marthi, B., Russell, S., & Wolfe, J. (2008). Angelic hierarchical planning: Optimal and online algorithms. *Proc. Int'l Conf. on Automated Planning and Scheduling* (pp. 222–231).

Molineaux, M., & Aha, D. W. (2014). Learning unknown event models. *AAAI*.

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

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

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

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

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

Ontanon, S., Synnaeve, G., Uriarte, A., Richoux, F., Churchill, D., & 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., Alford, R., Shivashankar, V., Leece, M., Gupta, S., & Aha, D. W. (2016). ACTORSIM: A toolkit for studying goal reasoning, planning, and acting. *Working notes of the Planning and Robotics (PlanRob) Workshop (ICAPS)*.

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

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

Shivashankar, V., Kuter, U., Nau, D., & Alford, R. (2012). A hierarchical goal-based formalism and algorithm for single-agent planning. *Proc. of AAMAS* (pp. 981–988). Int. Foundation for AAMAS.

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

Vallati, M., Chrpa, L., Grześ, M., McCluskey, T. L., Roberts, M., & Sanner, S. (2015). The 2014 international planning competition: Progress and trends. *AI Magazine*, *36(3)*, 90–98.

Vattam, S., & Aha, D. W. (2015). Case-based plan recognition under imperfect observability. *ICCBR*.

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

Wilson, M. A., Molineaux, M., & Aha, D. W. (2013). Domain-independent heuristics for goal formulation. *FLAIRS*.

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